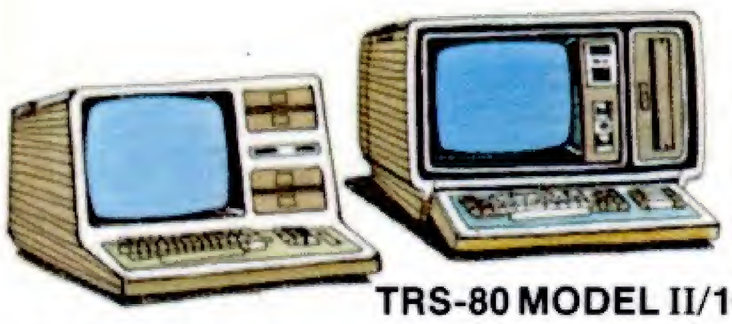
Radio Shack®

Catalog No. 62-2074

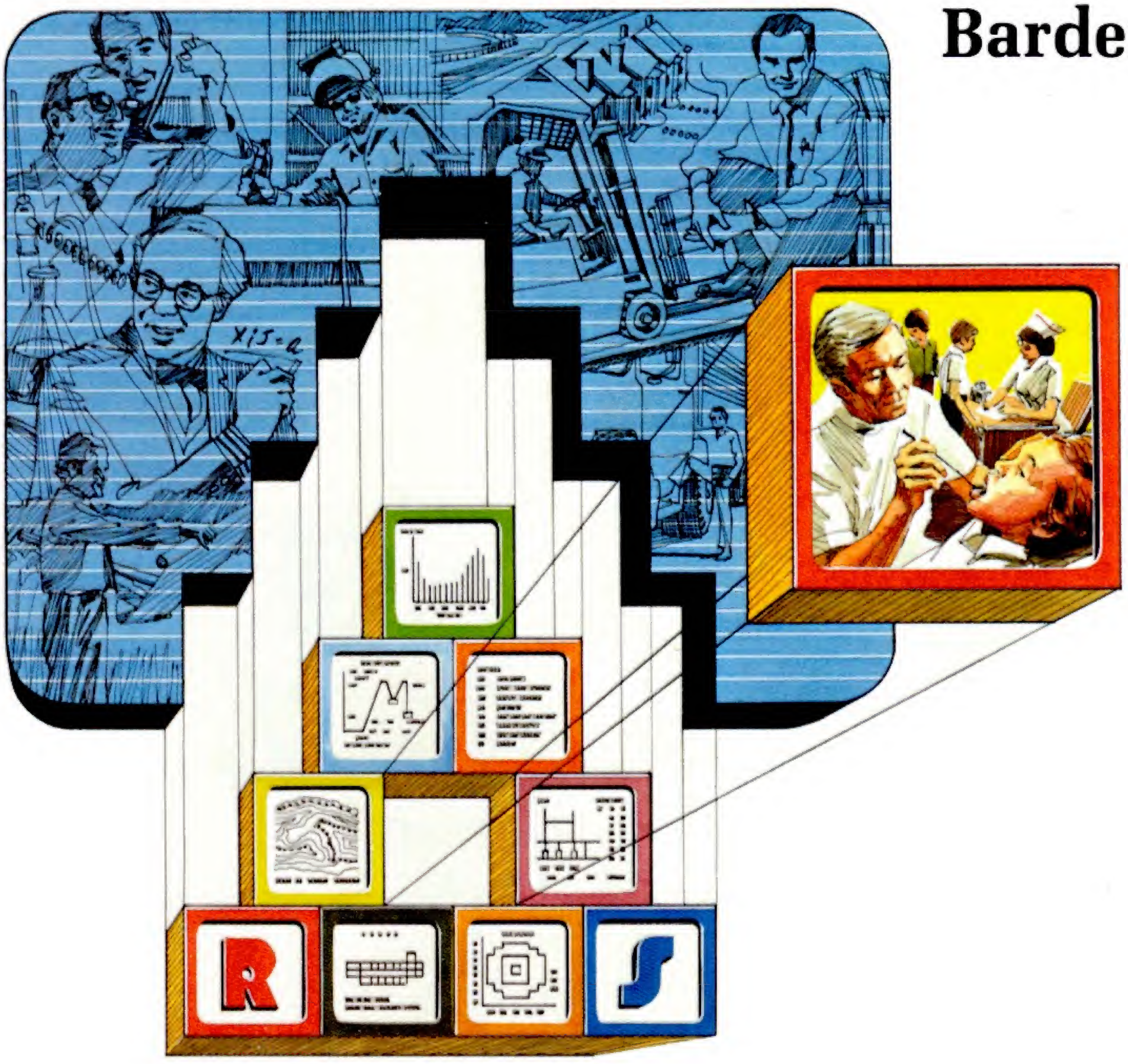


TRS-80 MODEL II/16

TRS-80 MODEL I/III

BUSINESS Programming Applications

William Barden



U.S.A. $4.95

BUSINESS APPLICATIONS PROGRAMMING GUIDE

by

William Barden Jr.

FIRST EDITION
FIRST PRINTING

International Standard Book Number:

Library of Congress Catalog Card Number:

Printed in the United States of America

Preface

"Is there a step-by-step procedure for writing business applications programs in BASIC on the Model I, II, or III?"

Radio Shack asked me this question some time ago in regard to a potential book idea.

"Sure," I answered, flippantly. "It would be about 4000 pages long with very fine print." However, I began pondering the question.

Now let's be honest . . . There's some learning involved in just finding out how to load programs into a microcomputer. There's a great deal more work in learning the BASIC language to write programs, and there's even more work in learning the programming craft — how to use the BASIC language to construct useful programs.

I knew I couldn't write a single book that would make it possible for every TRS-80 user to write large business applications programs after he had read it. Nobody could. But perhaps I could guide the user's efforts in the approach to learning how to write business applications programs. Maybe I could even give him some tools to use.

The result of this effort is *Business Applications Programming Guide.* It doesn't claim to be a step-by-step guide, but it does claim to provide you with a plan. It also gives you some powerful tools that you can use during the period when you don't know enough to easily write your own business programs. These tools are what I call the General Purpose Modules.

The General Purpose Modules are a collection of a dozen and a half BASIC programs that perform "difficult" functions for the first-time user — things such as loading and saving files on disk, sorting data, displaying forms on the screen, reading in keyboard data, and the like. They are grouped together in a core set of programs that you can use with additional BASIC programs to create business applications programs.

To show you how these modules work, I've included complete descriptions of the BASIC "code," implemented a complete Mail List program, and described an Information Retrieval Program and an Inventory System built around the modules.

By some study of the modules in this book, you will be able to do one of two things — you will probably be able to use all or some of the modules in constructing your own business applications and you may even be able to discard the modules and write your own programs from scratch. Either way, I will be delighted.

In addition to the General Purpose Modules — the "tools" — I've provided you with a general "blueprint" on how to proceed in business programming. Although I have strong opinions about some elements of programming, such as

good documentation, I've tried hard to make the plan a reasonable one. It is being used by the business programmers today that create good software and has been time-proven. The plan is nothing magical — it involves such things as flowcharting and design specifications — but you may never have actually seen the steps of the plan in print before!

I won't bore you with how wonderful microcomputers are for business applications and tell you that the TRS-80s are the best for your filling station or retail store. It is true, however, that we finally have reached a point where a small business user can spend a reasonable amount of money on a microcomputer system and use it to great advantage in virtually any business. This book attempts to show you how in a practical way.

There are four sections in the book.

Section I answers some basic questions about business applications and then outlines the steps of "the plan."

Section II provides an overall and then a detailed description of the General Purpose Modules. Each module is described almost down to a line by line basis. More importantly, general concepts for each module are discussed — file structures, sorting, and so forth.

Section III describes a typical business application that uses the General Purpose Modules, a MAILLIST program that will work on the TRS-80 Model I, II, or III. The complete program is contained in this book. The additional BASIC programs that use the General Purpose Modules are described in detail, along with the way the two sets of programs connect or "interface."

Section IV describes two other business applications in terms of use with the General Purpose Modules, an Information Retrieval program called KEYWORD, and an Inventory System called INVENT. Flowcharts and descriptive text are provided for both applications.

Two Appendices provide a shortened or "compressed" form of the General Purpose Modules, and a compressed form of the MAILLIST program.

I hope that you will use this book and the programs in it to advantage in your own business programs. The opportunities for "computerizing" a small business are there **now** — use them!

Business Applications Programming Guide
Table of Contents

Section I
Introduction to Business Applications Programming

Section II
General Purpose Modules

Section III
An Application Example

Section IV
Other Business Applications

Appendices

Section I
Introduction to Business Applications Programming

Chapter One
"But I've Had No Formal Training . . ."

Why is it so difficult to put together a business or other applications program? How long will it take? Can I do it even though I've had no formal training in computer programming? Will microcomputers **really** replace your tax man? We'll try to answer a lot of these questions in this chapter. We'll try to be brutally honest in our answers too. Microcomputers are a new technology and there's a "mystique" about them. It's sometimes difficult to cut through this mystique to obtain the solid answers that are required to make intelligent decisions in using microcomputer systems as effective business tools.

"All right, since you brought it up — can TRS-80s be used in business applications?"

As recently as 1977, some professional computer magazines declared that no microcomputers were being used in significant business applications. You'll still hear that refrain from larger computer manufacturers today, but the simple facts are that microcomputers such as the TRS-80 Models I, II, and III not only can do almost every job that larger computer systems can do, but can do them at a fraction of the cost.

"You hedged there — almost every job . . ."

There's no question that a very large system offers advantages over a microcomputer system — otherwise they could not compete. The advantages are greater storage capacity, ability to run many programs simultaneously, and higher speed. These features, however, are necessary only when the systems are being used by very large companies that maintain large programming staffs to write and maintain such programs as payroll for thousands of employees, "point-of-sale" terminals, or other large-scale operations.

"What kinds of applications can the TRS-80s handle?"

Payroll, accounts receivable, billing, accounts payable, job estimating, information retrieval, management reports, projections — virtually any business application that is now being done by manual methods can be implemented on a microcomputer.

"Can I find a program to perform my specific business application?"

One of the weak points in all microcomputer systems has been and continues to be, applications software. The reason is that good software takes a great deal of time to write. Because of the number of TRS-80 systems sold, there is more software available for them than any other **computer**, not just any other microcomputer. You'll find many common business applications from Radio Shack and other vendors. The more specific your requuirements, however, the more difficult it will be to find a program to do the job.

"What about buying programs from small vendors?"

Caveat emptor. There is a great deal of garbage being offered from many companies. Some of the programs simply don't work. Many of them don't work well. This is not to say that a small supplier won't have an excellent program for your application, or that a large supplier won't have garbage. Be very wary when purchasing any program. Try to find out specifically what it does, how fast it does it, and, probably the most important item, whether there is adequate documentation available.

"Ok then, how about writing my own business applications programs?"

Presumably that's why you've bought this book. It's certainly possible, even feasible. But there are some problems.

"What kind of problems?"

The problems generally fall into four areas — learning the system, learning BASIC, learning how to design and structure the program, and writing the program.

"Seems to be a lot of learning involved . . ."

Right. To say there isn't some hard work in learning computer basics, BASIC language, and programming techniques would be lying.

*"How **much** work?"*

Well, that depends on your aptitude towards computer systems. Some people soak the new technology up. Others never understand it. I remember that I was very reluctant to get involved with digital computers in the early sixties. They seemed very mysterious.

If you've got a reasonable head on your shoulders, there's absolutely no reason you shouldn't be able to learn all facets of computer systems in general and TRS-80s in particular. Hundreds of thousands of people have. And I mean people from every walk of life, not just those trained in mathematics, computer science, or engineering.

To get back to the amount of work . . . You must learn BASIC programming. BASIC is the simplest (but very powerful) high-level language usually used with the TRS-80s. You can get a good working knowledge of BASIC in as little as 20 hours. Typically, though, it'll probably take several months of part time study before you feel comfortable in BASIC.

Another area is learning about computer systems in general. You'll get some of this in the process of learning about BASIC. This will be an ongoing process, however, as the technology changes, so it's hard to put a time value on it.

The last area of learning involves learning about the design and structure of BASIC programs. That's what this book is all about. If you have some

knowledge of BASIC, then we'll try to sharpen that knowledge up with this book and show you how to use your BASIC expertise in constructing useful business applications programs.

"Will this book teach me BASIC?"

No, sorry. As I say, it will "sharpen up" your BASIC. We'll be explaining "tricky" and some not so tricky BASIC statements. Before you can use this book, however, you must have some experience in using either TRS-80 Model I Level II BASIC, TRS-80 Model III Level III BASIC, or TRS-80 Model II BASIC. You don't have to be an expert BASIC programmer before reading this book; you should be familiar enough with the language to recognize most of the commands and to have used many of them in short practice code of your own. Use your particular BASIC Reference manual as that — a reference manual — if you encounter difficult commands in the BASIC code in this text.

"What about Level I BASIC — can I use that for programs?"

Level I on the Model I or III is great for learning BASIC and for simple games and applications programs. It is not an option for business programs, however. The Level II or III upgrade is an absolute requirement for business applications, and we won't be considering Level I BASIC in this book.

"What about the fourth problem you mentioned earlier, writing the programs?"

This book should certainly help in writing your own BASIC application programs. Much of the hard work is in structuring the program — dividing it into convenient segments of BASIC code that perform specific functions. We've provided some "standard" short programs that you can use as building blocks in constructing your own applications.

"What about other languages such as FORTRAN or COBOL?"

FORTRAN (FORmula TRANslator) is primarily a scientific language useful for programming mathematical and engineering computations. COBOL (Common Business Oriented Language) is indeed a business language but is perhaps best suited for large-scale systems or for companies that can utilize a staff of existing COBOL programmers. BASIC remains one of the easiest languages to learn and is probably best for highly "interactive" systems such as the TRS-80 Models I, II, or III.

"How long does it take to write a BASIC program?"

Take a look at some typical "code" from a BASIC program listing in Figure 1-1. This is code of average complexity. Each line generally has one BASIC command or function on it. Typical time spent to code this program was about 60 lines per day. And don't forget, we're talking about a trained computer professional under the best circumstances and moderately hard "code."

```
11110 XX=0
11120 PRINT @ YE,XB$+"                    ";
11130 IF XB=3 GOTO 11280
11140 XC$=""
11150    XI$=INKEY$:IF XI$="" GOTO 11150
11160    IF XI$>CHR$(YK) GOTO 11200
11170    IF XI$<>CHR$(YK) GOTO 11190
11180    XX=1: GOTO 11290
11190    IF XI$=CHR$(13) GOTO 11230
11200    XC$=XC$+XI$
11210    PRINT @ YE+LEN(XB$)+1,XC$;
11220    GOTO 11150
11230 IF XB=0 THEN XC=VAL(XC$)
11240 IF XB<>2 GOTO 11290
11250 IF XC$<>"YES" AND XC$<>"Y" AND XC$<>"NO" AND XC$<>"N" GOTO 11120
11260 XC$=LEFT$(XC$,1)
11270 GOTO 11290
11280    FOR XI=1 TO 900:NEXT XI
11290 RETURN
```

Figure 1-1. Typical BASIC Applications Code

"That's not very fast. I spoke with one programmer who said he could code my billing system in a week . . ."

Beware of such estimates from people whose business cards read "John Smith, Computer Programer (sic)." He either wants your business very badly, doesn't have much experience, or is going to provide a totally inadequate package that will have to be constantly changed and updated and will have no instructions on use. Common sense tells us that just preparing instructions on how to use such a system will take three or four days.

By the way, when I said 60 lines per day, I meant 60 lines per day on the average. During the **design** phase of a program, you'll be doing a lot of head scratching with no coding. During the **coding** phase you'll probably be "laying down" code at a faster rate. The 60 lines per day is just a nominal figure useful in estimating times to code a specific job; typically computer software companies would use such standards in determining schedules for software.

"How big is a typical program?"

Most BASIC programs are "compressed" so that they make better use of memory, as shown in Figure 1-2. When this is done, the typical size ranges from 200 lines up to 1000 or more. In this case, figure two "normal" commands per line, 30 lines per day, or 10 to 50 days or more to implement the program.

```
11110 XX=0
11120 PRINT@YE,XB$+"                    ";:IFXB=3GOTO11280
11140 XC$=""
11150 XI$=INKEY$:IFXI$=""GOTO 11150
11160 IFXI$>CHR$(YK)GOTO11200
11170 IFXI$<>CHR$(YK)GOTO11190
11180 XX=1:GOTO11290
11190 IFXI$=CHR$(13)GOTO11230
11200 XC$=XC$+XI$:PRINT@YE+LEN(XB$)+1,XC$;:GOTO11150
11230 IFXB=0THENXC=VAL(XC$)
11240 IFXB<>2GOTO11290
11250 IFXC$<>"YES"ANDXC$<>"Y"ANDXC$<>"NO"ANDXC$<>"N"GOTO11120
11260 XC$=LEFT$(XC$,1):GOTO11290
11280 FORXI=1TO900:NEXTXI
11290 RETURN
```

Figure 1-2. "Compressed" BASIC Program

But 30 into 200 is 6 days and 30 into 1000 is 33 days . . .

The increased time of the smaller program represents the larger proportion of "paperwork"; the increased time of the larger program represents the larger proportion of design and debugging time for a large program. Figure 1-3 shows a rough graph of times to program an application. Times tend to flatten out on the smaller program end and increase on the larger program end.

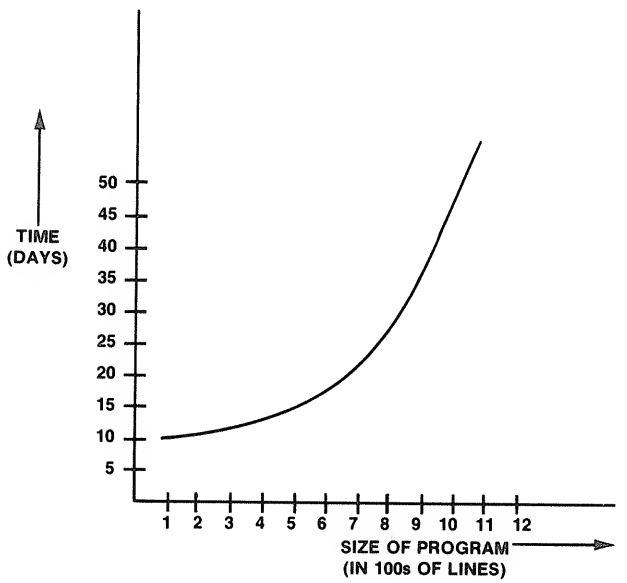


Figure 1-3. Programming Time Estimates

"Can I hold you to these standards?"

Absolutely not. There are too many variables — whether you have a cassette or

disk system, how experienced you are, whether you have system problems, how often you make backups, and so forth. These are guidelines only and represent the best case. A good rule of thumb to follow is that programs always take longer than expected!

"Gulp. Thirty days is a lot of time to write a program . . ."

Yes, but it's realistic. Most people don't realize the amount of work involved in writing software. That's why custom-tailored programs are expensive.

"Isn't there an easier way to get a custom-tailored program for my application?"

It might be possible to take an existing package and modify it for your needs. However, in a lot of cases this means learning how the entire program works, and there's that investment in time again. This is hard especially when the commands are compressed together in one line, blanks are deleted, and there aren't any explanatory REMarks. In some cases it may be almost impossible.

Another way to cut down the time for designing and programming is to use a base of "standard" code for specific functions that come up in every program — inputting text data, displaying "forms," and so forth. We've got about a dozen and a half modules — the General Purpose Modules — that are guaranteed to work in the book. You can use these, or build up a "library" of your own.

As you might guess, it gets easier and easier to design and code the more that you do. You do have the advantage of being in complete control of the software, also.

"How does a BASIC business program differ from other BASIC programs?"

We've used the term "business applications" rather loosely here. What we really mean is any applications program that is dedicated to implementing typical business functions such as inventory, accounting, record keeping, filing of data, and so forth. These functions have some things in common. One of the things they have in common is that they are all oriented towards handling collections of data - records of transactions. Another characteristic they share is that they are not concerned with a great deal of "number crunching." Stock market simulations and economy models notwithstanding, most business programs do simple arithmetic computations. A third characteristic is that they are "report-oriented"; the final result is a report on a display or line printer of an updated inventory file or mail list or other easily interpreted collection of data. The same thing holds true for input; input is usually textual data that is in conveniently interpreted form.

Some of the things we won't be talking about in this book are games, high-speed graphics, assembly-language, or other esoteric topics.

"Is there a step-by-step procedure to follow to write a BASIC applications program?"

If there is, please call me collect. But seriously . . . What we'll do here is to explain what we feel is a step-by-step procedure to follow. The process we'll describe here will be one that professional business programmers follow, suitably modified for the TRS-80 systems. It will be oriented to keep a beginning programmer out of trouble. Various BASIC programmers may scoff at some of the time-consuming aspects of this procedure, but it is our sincere belief that the programs written by such a procedure will be less prone to failure, faster in execution, and ultimately take less time to develop and maintain than the ones generated by sitting down at the console without any planning. We'll give you the steps in the next chapter.

Chapter Two
Steps in Developing an Applications Program

We're going to outline a step-by-step procedure for developing business-oriented applications programs in this chapter. The procedure we're presenting is not one that we just created for this book. The basic elements of the procedure have been used by good programmers for years in writing applications programs. Because there are some differences in the operation of the large computer systems on which the procedure is based, we've modified the steps to work well with microcomputers such as the TRS-80 Models I, II, and III.

Some of the steps will tend to be somewhat subjective — programmers may differ in their opinions about certain aspects of the procedure. When this is the case, we'll let you know the options and recommend a course of action. Later, after you've developed a number of applications programs, you'll form your own ideas about the most effective way to produce programs.

The Basic Plan

The basic steps in developing applications programs are shown in Figure 2-1 in "flow chart" form. As you know from reading other TRS-80 manuals and computer books and magazines, flow charts are used to show the flow of a procedure. We'll be using a few very simple flow chart symbols throughout the book; they are shown in the figure. We'll talk more about the use of flow chart symbols later in this chapter.

The steps from the figure are these:

1. Learn the characteristics of the system
2. Learn the BASIC you'll be using
3. Research the applications problem
4. Write a design specification
5. General program design
6. Flowchart the program
7. Code the program
8. Enter the program
9. Debug the program
10. Create the final version of the program
11. Write an operational manual for the program

Note that some of the "flow" in this procedure is repetitive, or what computer people call "iterative." There may be one or more iterations or repeats of certain sections of the procedure. We'll talk about that in a while.

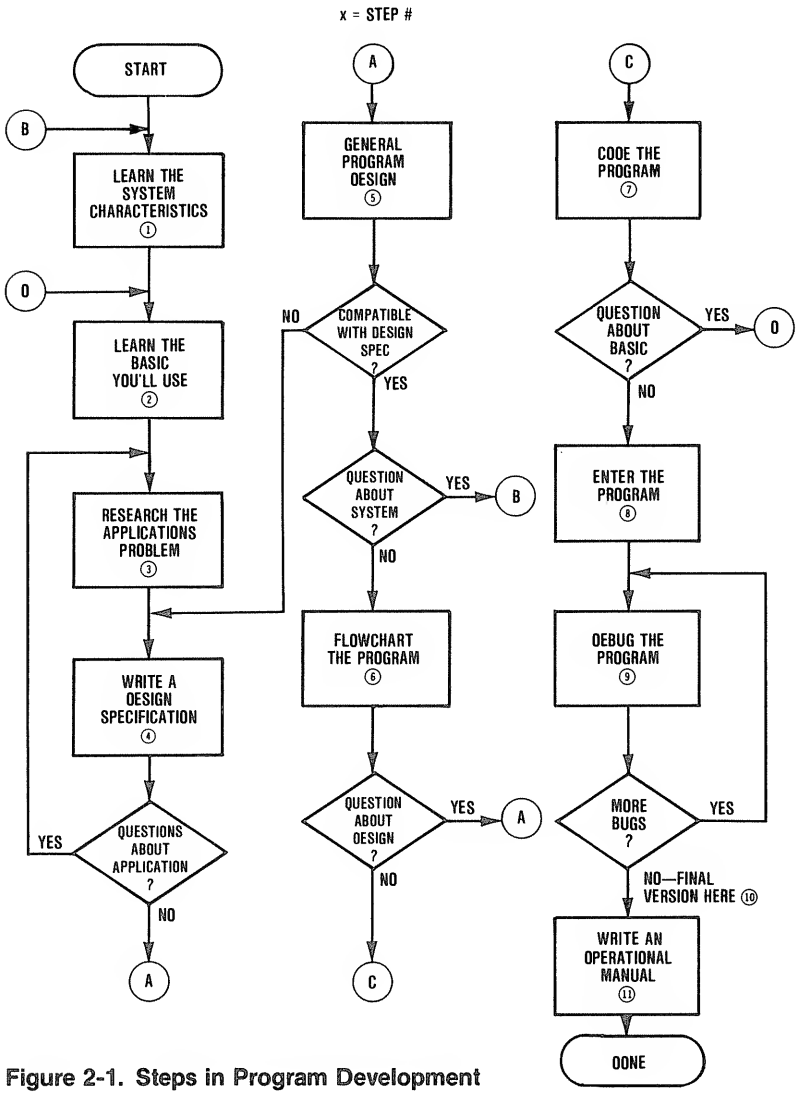


Figure 2-1. Steps in Program Development

Step Number One: Learn the Characteristics of the System

The first step in the procedure is to learn everything possible about the system you'll be using. The reasons for this are obvious. You may have created an excellent BASIC program to provide automatic billing. The program may be error free. When it comes time to run the program, however, you may find that you simply don't have enough memory to hold the program and the data files you will be working with. Or the line printer may not be able to print your billing information on the forms you require. Or the clerk that inputs the billing

information may not know what to do when a disk error occurs and runs off with the office leasing agent . . .

Of course it's not necessary to delve into the schematic diagrams of the system to find out how the TRS-80 operates electrically. But it's good to get an idea of the general operation of the system in terms of the memory layout of RAM, ROM and video; to find out the general layout of disk files, and to find out specifics on I/O devices such as line printers.

You've already learned a significant amount about computers to reach this point, but don't stop here. Continue to learn about microcomputers in general and your TRS-80 system in particular. That's why we've shown the path back to this step in Figure 2-1. A gross misunderstanding about disk or memory capacity or printer speed may require some additional research into the system!

Step Number Two: Learn the BASIC You'll be Using

We mentioned in Chapter One that you should have some knowledge of Level II, Level III, or Model II BASIC at this point. The BASIC code in this book can be used to clarify certain points about the use of BASIC commands. You should become very familiar with most basic commands and functions, however. Not knowing that you can easily convert from the string "123.56" to a value of "123.56" might result in a lot of laborious coding where a single function would suffice!

This is one of the major obstacles that all programmers, professional or part time, face — becoming familiar enough with the language so that the commands come instantly to mind. To a certain extent, the language determines the **structure** of the program, so it's very important to think in terms of BASIC when designing a large program.

Table 2-1 shows a recommended "priority" of study for BASIC commands and functions in business applications programs which might be a help in determining where to expend the most effort.

Here again as shown in Figure 2-1, learning BASIC will be an iterative process, at least until you've written enough code so that you hardly ever have to refer back to the BASIC manual.

High Priority:

CLEAR, CLOAD, CLOSE, CLS, CMD''T'', CONT, CSAVE, DATA, DEFINT, DEFSNG, DELETE, DIM, EDIT, END, FOR..TO..STEP/NEXT, GOSUB, GOTO, IF..THEN..ELSE, INPUT, INPUT#b, KILL, LIST, LLIST, LOAD, LPRINT, LPRINT TAB, LPRINT USING, MERGE, MID$, NAME, NEW, OPEN, PRINT, PRINT @, PRINT TAB, PRINT USING, READ, REM, RESTOR, RETURN, RUN, SAVE, STOP

INSTR, LEFT$, LEN, MEM, MID$, RIGHT$, STR$, STRING$, VAL

Next Priority:

CLOAD?, CMD''R'', CMD''S'', DEFDBL, DEFFN, DEFUSR, ERROR, FIELD, GET, INPUT#-b, LINE INPUT, LINE INPUT#b, LSET, ON ERROR GOTO, ON..GOSUB, ON..GOTO, POKE, PRINT #-b, PRINT #b, PUT, RESET, RESUME, RSET, SET, SYSTEM, TRON, TROFF

ABS, ASC, EOF, ERL, ERR, FRE, INKEY$, INT, LOF, MKD$, MKI$, MKS$, PEEK

Some Never Used, Some Sophisticated:

CMD''I'', CDBL, CHR$, CINT, CSNG, CVD, CVI, CVS, FIX, POS, RND, SQR, TIME$, USR, USRn, VARPTR

Seldom Used:

ATN, CMD''D'', COS, LET, OUT, EXP, INP, LOG, POINT, RANDOM, SGN, SIN, TAN

Table 2-1. Study Priority of BASIC Commands

Step Three: Research the Applications Problem

This is the step in the procedure that you are probably most familiar with. In many cases you are an expert in the application itself. You know how many accounts you have for billing, what types of personnel records you are keeping, or the steps in estimating costs for an air-conditioning installation.

This step is probably the single most important step of the whole process. In it you will have to convert the manual methods that you are using into computer system equivalents. To do that you will have to define the "dimensions" or "parameters" of the problem. How many accounts will the program have to handle? What type of information is held in each account? How many characters will have to be held in each field of information? What special codes are used to represent different billing information? What provisions should be made for future expansion? How often should the "file" be updated?

Although the manual methods seem obvious to you and the people that work on the application, are they? The computer has no "common sense." It must follow a specific sequence of operations. At this point an attempt should be made to define the manual operations by writing down the "flow" with contingencies for every condition. This will help define the "common sense" approach that you may now be following in your application. In fact, this writing process is absolutely necessary in converting to a computerized process.

A sample analysis of a portion of an order entry application is shown in Figure 2-2. Note that the analysis doesn't have to be a flowchart (it could very well be), but can be a simple handwritten account of the manual process. Here again this may require one or more "iterations" to clarify what really goes on in the manual process. You may even find that nobody really knows what **does** go on in the operation!

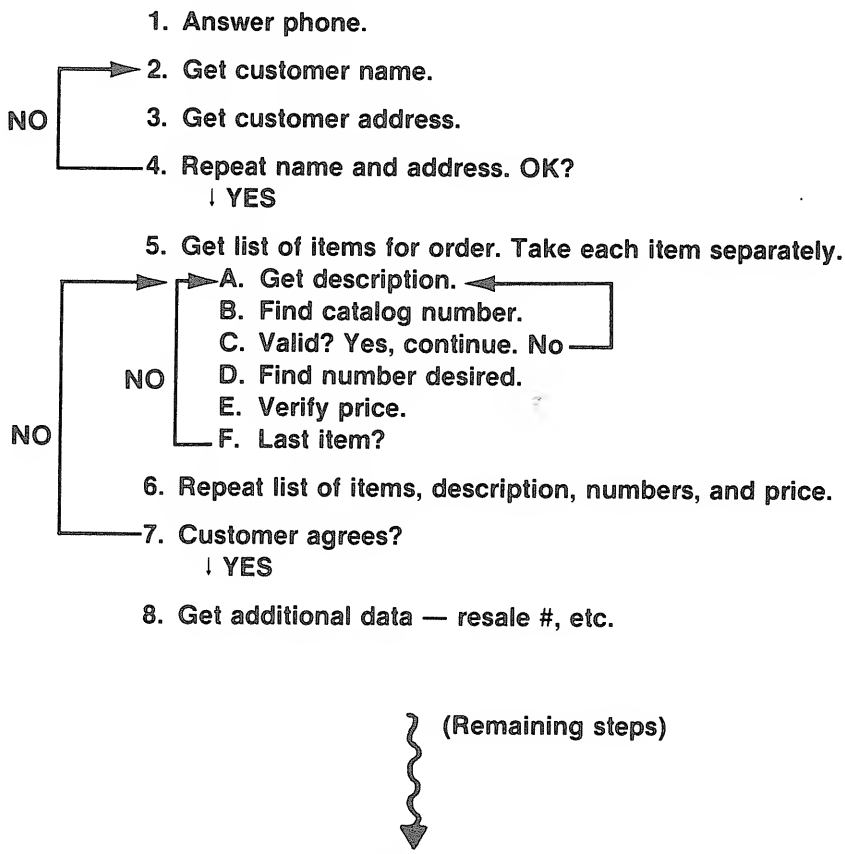


Figure 2-2. Order Entry Analysis Example

Step Four: Write a Design Specification

The next two steps, writing the design specification and designing the program, may be on the same level, or one may precede the other. Some "systems analysts" or programmers would be in favor of writing a design specification first before program design. This is known as "top down" design. Others would be in favor of doing both at the same time. Still others would produce the design spec after the program design.

We favor, whenever possible, writing a design specification with **operational procedures** first, and then producing a program design. The design spec would consist in this case of a set of written procedures on how to use the system. It would show all screen and line printer forms and list the various functions available. A page of a design spec for a program developed later in the book is shown in Figure 2-3.

MAILLIST Design Specification

OVERVIEW:

MAILLIST is a complete mailing list program designed to operate on the TRS-80 Model I, II, or II Microcomputer Systems.

It can handle up to several hundred entries of forty or fifty characters each, or more entries if the entry size is reduced. MAILLIST builds an in-memory file of mailing list entries. Entries can be added, deleted, or modified at any time. High-speed "list" sorting is used to alphabetize each entry as it is added.

Entries may be displayed on the screen one at a time or in user-specified ranges. Entries may be printed on the system line printer in mailing label format or in any user-defined format, such as a line by line report.

Any character string, from a single character to larger strings, may be used in a "search" mode. MAILLIST will search all entries for the given string. This feature can be used to locate a street address, zip code, or other data.

Entries are arranged in alphanumeric order based upon the entire entry. JONES,ED,25555 OAKHURST appears before JONES,ED,25556 OAK-HURST, for example. The "sort key" is essentially the last name. At any time the list of entries may be resorted based on any other field, such as zip code. This secondary sort can then be displayed or printed. This feature is handy for arranging the entries in zip code order for bulk mailing requirements, for example.

The complete list of entries may be saved as a disk or cassette (Model I/III) file with a user-specified name. Files can be loaded from disk, or two files may be merged together into a new "master file," again with user-specified name.

MAILLIST is "menu-driven" and uses a complete set of "fill-in forms" to prompt the user. Complete editing features are provided for entry of user data.

LOADING PROCEDURE:

To run MAILLIST, first load BASIC. If you are using the Model II, specify BASIC -F:1 to allow you to have at least one disk file. Next, load

Figure 2-3. Sample Design Specification

How closely will the design spec match the manual methods in Step 4? In some cases it will be possible to closely emulate the manual methods in the computer system; in other cases the manual methods simply don't lend themselves to being duplicated in the computer. In most cases, some changes to the established

manual processes will have to be made. Manual methods allow for street addresses of virtually any length, for example. In the computer version some limit must be put on the number of characters for the street address. Manual methods allow for flagging certain accounts with red tags; in a computer these flags must be special codes that require a search.

How closely will the operational portion of the design spec match the final program? This is related to programmer experience and familiarity with the system and BASIC. For the first several applications, there may be some misconceptions about the system (Step 1) that result in changes to the design spec. After several larger programs have been implemented, the operational portion of the design spec should be very close to the actual operation of the program.

The following items should be in the design spec:

- Overview of what the application does
- Loading procedure
- How to start execution
- List of program functions
- Description of each separate program function
- Screen appearance of "menus"
- Screen appearance of input forms or "prompting" for operator input
- Screen appearance for reports or display of records
- Line printer appearance for reports

(One cautionary rule about design specs: Never start your spec with the words "Welcome to the Wonderful World of Acme Business Software . . . .")

Step Number 5: General Program Design

The next step is to decide how to design and **structure** the program. In some cases this step precedes the design spec or operates in conjunction with it. In this phase you will have to ask yourself such questions as:

- How will I store text and data in the program?
- Should I use arrays and, if so, how large will they be?
- What kind of searching algorithms do I need to locate the separate data items?
- What about disk storage? Should I use random or sequential files?
- How should I structure the program — should it be one massive set of code, or should I divide it up into **modules**?
- What variables should I use, and what **data types** should they be?

Much of this design is going to come from experience. You may implement a design by using a string array of items and find that searching for a certain item takes several minutes. On the next application you may discard that method as too time consuming and look for other approaches.

Because this step is so much related to experience, we've made it a lot simpler. We've defined a set of general purpose **modules** that have a kind of "built-in" structure, one that uses string arrays geared to high-speed merging of data. We don't claim that the approach is best for every application. It is an approach that will serve you until you can define your own program structure, however.

We'll be discussing the basic design and modules in the next section.

Step 6: Flowchart the Program

The next step is to flow chart the program. There are many programs that are never flowcharted, especially those sold for microcomputers in BASIC. Is flowcharting necessary? Or is it simply possible to sit down at the TRS-80 and enter BASIC code?

It's certainly possible to enter BASIC code for small programs without using flowcharts and not get into trouble. BASIC on the TRS-80s is very "interactive," and it is easy to "edit" and modify BASIC commands and functions until the code works.

The larger the program, however, the more difficult it is to take advantage of this interactive nature. Unless program plannning is done, the end result of entering code and modifying it until it works is a hodge-podge of statements with no logical flow at all. An old programming joke talks about the two methods of developing a program. One method is to sit down and carefully plan out the program by a design spec and flowcharts. The other method is to go to a wastebasket in the computer room, pick out **any** listing, and modify it until it works the way you want it to! Entering code without planning results in programs that look similar to the second method.

Why use flow charts at all? Why not simply jot down notes on the flow of the program? Some programmers do this and it works well for them. However, we'd strongly recommend that you flowchart at least for the first two or three large programs. Then, if you find that an alternative method works (and your program structure is clean), use that method.

We've included flow charts for many of the application examples used with the General Purpose Modules. The flow charts are not descriptive down to a command level, but simply provide a general "flow" of the program. A sample is shown in Figure 2-4.

We've used only seven symbols in our flowcharts. The rectangle (▭) is a "procedure box." Notes inside the box describe processing ranging from incrementing a counter to scanning an array for an entry. The statement number for the approximate point in the program where the processing is done is sometimes placed at the top left of the symbol.

The diamond (◇) is a "decision" symbol. Two or more paths may be taken from

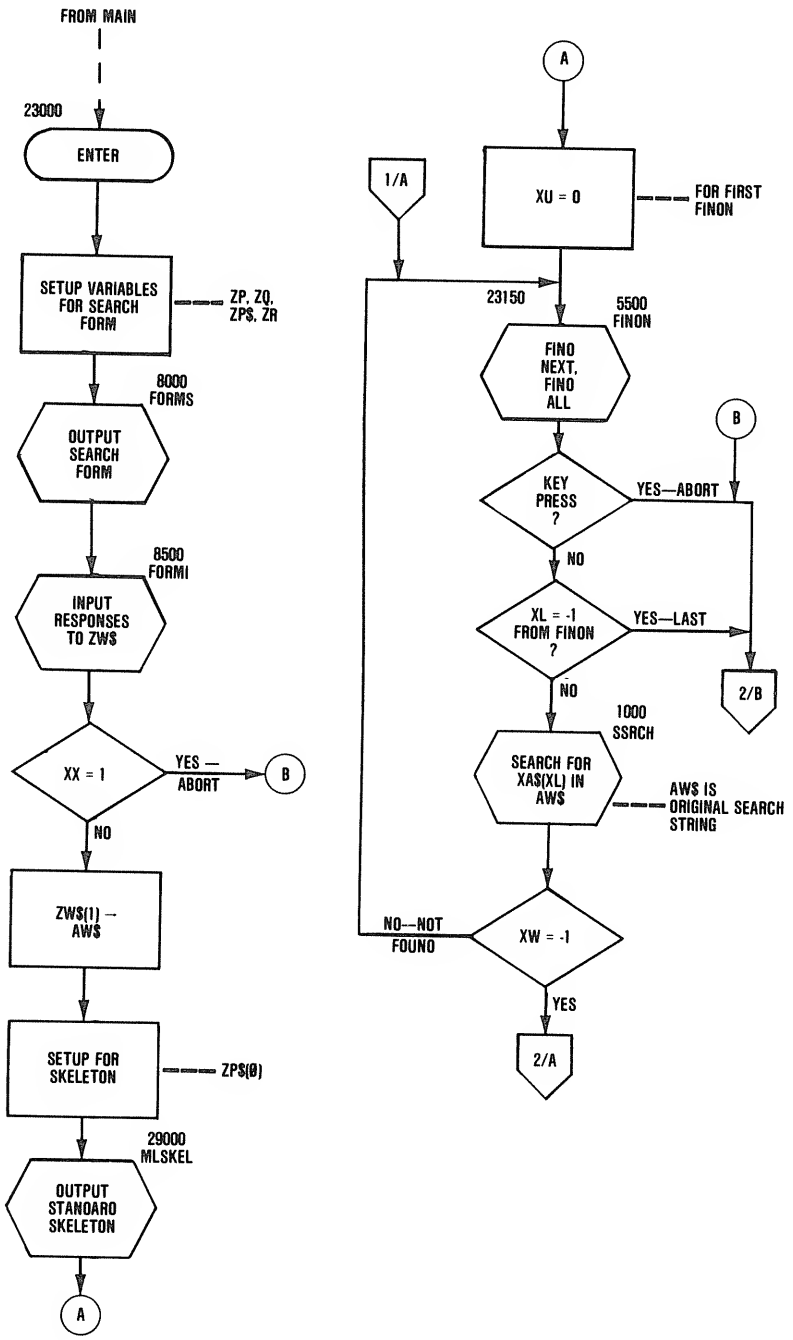


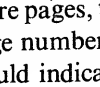
Figure 2-4. Sample Flow Chart

the decision point. An optional number at the top left is the statement number at which the decision is made.

The  symbol indicates that a **subroutine** is executed. The name of the subroutine is on the top right of the symbol. An optional statement number for the point at which the GOSUB takes place is on the top left of the symbol.

The oval () indicates a starting or ending condition. The triangle (▽) indicates a "branch out" or a RETURN.

The small circle () is an "on-page" connector. It contains a letter indicating where on the same page the path is completed.

The  symbol is an "off-page" connector. When the flowchart is on two or more pages, this symbol is used to connect the path from one page to another. A page number with a letter indicates the connection point.  , for example, would indicate "page 4, point A."

Note that the flow is generally down and to the right. As you can see from Figure 2-1, flowcharting is an iterative procedure. The flowcharts may have to be redone, or may even result in modifications to the general program design (step 5), or design spec (step 4).

Step Number 7: Code the Program

Once the flowcharts have been developed, it's easy to code the program. Can you now sit down at the keyboard and start "laying down code"? Here again, programmers do it, and some do it very well. For your first several large programs, however, it's advisable to write down the "rough" code using pencil and paper, leaving plenty of space between statement lines.

In some installations, all programming is done on "coding sheets" which are then submitted to a "data entry" clerk who enters the code into a program file. This file is then "compiled," and the programmer gets the listing to "desk check," or look over for errors. One of the chief reasons for coding sheets is that large systems do not allow the programmer the luxury of being able to use the system on an individual basis — many "jobs" are run simultaneously. On the TRS-80s, however, we have a single-user capability. Still, some further thought is needed before sitting down to enter the program. Coding sheets aren't required, but some paper and pencil work is.

Step 8: Enter the Program

We've finally made it to the keyboard! At this point the BASIC program can be entered, using your code from paper, or your flowcharts, or a combination of both. What does the code look like? (Please, no derogatory comments . . .)

There are two main criteria to be considered in BASIC programs on the TRS-80. The first is readability and format. The second is execution speed.

BASIC is sometimes said to be self-documenting. In other words, you can pick up a BASIC listing, look at it and tell immediately what is happening in the program. A good example of this is shown in Figure 2-5, where it's immediately obvious what the program is used for!

```
11000 GOTO11110'PROMPT
11110 XX=0
11120 PRINT@YE,XB$+"                    ";
11130 IFXB=3GOTO11280
11140 XC$=""
11150 XI$=INKEY$:IFXI$=""GOTO11150
11160 IFXI$>CHR$(YK)GOTO11200
11170 IFXI$<>CHR$(YK)GOTO11190
11180 XX=1:GOTO11290
11190 IFXI$=CHR$(13)GOTO11230
11200 XC$=XC$+XI$
11210 PRINT@YE+LEN(XB$)+1,XC$;
11220 GOTO11150
11230 IFXB=0THENXC=VAL(XC$)
11240 IFXB<>2GOTO11290
11250 IFXC$<>"YES"ANDXC$<>"Y"ANDXC$<>"NO"ANDXC$<>"N"GOTO11120
11260 XC$=LEFT$(XC$,1)
11270 GOTO11290
11280 FORXI=1TO900:NEXTXI
11290 RETURN
```

Figure 2-5. Self-Documentation (Tongue in Cheek)

Not so obvious, is it? Contrast this with the code from Figure 2-6. Figure 2-6 is "formatted" with plenty of REMarks, indentations to show loops, and single commands (for the most part) on each line.

Which is best?

The answer is that the code shown in Figure 2-5 is best for memory storage requirements and execution speed and worse for debugging and readability. The code shown in Figure 2-6 is worse for memory storage and execution speed and best for debugging and readability. The difference in execution speed between the "compressed" code and "pretty format" code may be 2 to 1. On the other hand, it may take four times as long to debug the compressed code because it's so difficult to understand what is happening. Which should be used?

The approach we'll take is that REMarks, formatting, and single statement lines will be used for entering and debugging the program. Once the program has been fully debugged and tested, we can then delete all REMark lines, delete all spaces between commands, and put multiple statements on single lines. Most of the BASIC code shown in this book will be in the "pretty format" structure for explanation. We've also included a compressed version for high-speed execution and less memory storage (see Appendices I and II).

```
11000 GOTO 11110 'PROMPT
11010 '*****************************************************
11020 ' THIS IS THE "PROMPT" MODULE. IT OUTPUTS A GIVEN MESSAGE
11030 ' AT LAST LINE AND READS IN A USER STRING OR NUMERIC
11040 ' RESPONSE.
11050 '      INPUT: XB$=MESSAGE TO BE OUTPUT
11060 '             XB=0 IF NUMERIC RESPONSE,=1 IF STRING,
11070 '               =2 IF YES OR NO RESPONSE,=3 NO RESPONSE
11080 '      OUTPUT:XC$=STRING RESPONSE OR "Y" OR "N"
11090 '             XC=NUMERIC RESPONSE OR 0 IF ENTER
11100 '*****************************************************
11110 XX=0
11120 PRINT @ YE,XB$+"                    ";
11130 IF XB=3 GOTO 11280
11140 XC$=""
11150    XI$=INKEY$:IF XI$="" GOTO 11150
11160    IF XI$>CHR$(YK) GOTO 11200
11170    IF XI$<>CHR$(YK) GOTO 11190
11180    XX=1: GOTO 11290
11190    IF XI$=CHR$(13) GOTO 11230
11200    XC$=XC$+XI$
11210    PRINT @ YE+LEN(XB$)+1,XC$;
11220    GOTO 11150
11230 IF XB=0 THEN XC=VAL(XC$)
11240 IF XB<>2 GOTO 11290
11250 IF XC$<>"YES" AND XC$<>"Y" AND XC$<>"NO" AND XC$<>"N" GOTO 11120
11260 XC$=LEFT$(XC$,1)
11270 GOTO 11290
11280    FOR XI=1 TO 900:NEXT XI
11290 RETURN
```

Figure 2-6. Formatted BASIC Listing

You should probably use many REMarks thrown in at appropriate places and logically grouped statements in a single line from the start. The REMarks should contain truly useful information. It's considered bad form to put in a REMark such as ''1000 REM THIS IS A REMARK.''

After you've debugged and tested your program, you can then delete the REMark lines and "renumber." In later programs, if you find that you can easily read BASIC code that is run together, by all means use it, but bear in mind that the saving in memory space and execution speed between a program with 60-character wide lines and a program with 20-character wide lines is only about 15%.

Step 9: Debug the Program

All right, you've entered the program and stored it as a file on disk or cassette. What's next?

The next step, of course, is debugging — running the program and correcting any "logic" or minor errors that have occurred. Logic errors pertain to program design. If you have done a good job on Steps 5 and 6, there should be a minimal number of major logic errors in the program. There will probably be a huge number of other errors, however. Don't be dismayed — it happens in every

program. Sure, in speaking to some programmers it sounds as if they spew out perfect code every time. Let's just say they are stretching the truth somewhat.

Faced with this superabundance of errors, how do you logically proceed? Most programmers just plunge right in and start correcting each error. This is where microcomputers that use interactive software such as the TRS-80s are extremely powerful. It's easy to find the error, correct it by a line Edit, run the corrected version immediately, and test it again.

After a half dozen or so errors, save the corrected program on disk or cassette, and keep on debugging. Eventually you will reach a point where the number of errors has been reduced to a manageable number. At this point you can write down a list of the more subtle bugs and work on them one at a time. At this point too, you should think about saving backup versions of your program, so that you don't inadvertantly wipe out all of your corrections with no record of what they were.

When every last bug has been found and stomped, the debugging work has just begun. Too many programs are now deemed "checked out" and released. What **should be done**, and what is usually done in professional software houses, is to now develop a "test plan" to **exercise** the program. Write down a list of features that should be checked out. For example, if the program should be able to handle 100 accounts, try it with 100 accounts. If a name field should be able to handle 1 to 23 characters, try names of 1, 23, and 5 characters. If allowable menu values are 1 through 10, try 0, -1, 1, 10, and 11.

Look at the program from the point of view of your most critical friend (or enemy) and try to develop a list of several dozen, or a hundred, or several hundred test cases. Incorporate these into a test file on disk or cassette if possible. Keep on debugging the program until the list runs perfectly.

Isn't this time consuming? Yes, definitely. In some cases, testing and exercising the program may take 30 to 50% of the total program development time. However, industries find that program "maintenance" costs — finding and correcting errors in existing programs — **far exceed** the costs of program development!

Step 10: Create the Final Version of the Program

At this point you've debugged and tested the program. If the program has replaced an existing manual system, the program has probably been run "in parallel" with manual methods until every last flaw has been found and corrected. When you are confident that the program is working well, **and** you have a clean copy of the final version with backups safely stored on disk, **and** you have a number of program listings that **exactly correspond to the version of the program that is running**, you're almost done.

If the program operates at a fast enough speed and does not take up too much memory, you **are** done except for Step number 11. Leave the program as it is. If you'd like to increase its speed and reduce its memory requirements, then you may clean it up by carefully deleting any remaining REMark lines, deleting spaces between commands, and merging statements in single lines. Invariably, however, this process is going to introduce new errors. To detect these new errors, you've got to repeat the test plan once again, and even run the system in parallel. It is your decision at this point as to whether the extra debugging time is worth the increased speed and reduced memory requirements.

There is an old programming axiom (circa 1945) that "there are no final versions of programs." This is largely true. There are probably still bugs in your program design or code. Be aware that they may still show up months later, and be prepared by always keeping an up-to-date version of the program listing and notes on execution.

Step 11: Write An Operational Manual for the System

You're almost done. If you're lucky, the design spec is still valid as far as program operation. Usually, some minor corrections are necessary. Add the corrections and use it as an operational manual for the program. Too many programs have inadequate documentation. They may be excellent programs, but totally unusable by inexperienced personnel.

If you are to be the sole user of your program, it's still a valid idea to have an operations manual. It's very easy to be confused by what you did in a program that you wrote six months previously. (Occasionally you'll discover that the programmer you've condemned with an **expletive deleted** is yourself!) If other people are to use your program, then a detailed operational manual is absolutely necessary.

Budget Time for Program Development

We mentioned previously that an average of 30 lines per day was typical for program development of a moderately hard, moderately long business applications program. In other words, a 600 line program would take 20 8-hour days to develop. How does this time break down into the various steps we've outlined here?

Figure 2-7 shows a typical "pie" for the different segments of steps 3 through 11. This is meant to be a guide only, and not a hard, fast analysis. (Many books have been written about program development budgeting. Many managers have met their fate after the 32nd schedule slip.)

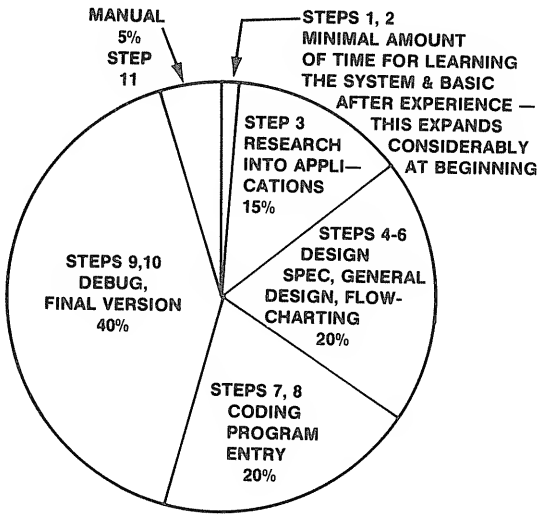


Figure 2-7. Program Development Breakdown by Task

Using the General Purpose Modules in This Book

By providing you with several dozen and a half well-documented, checked-out programs that handle form output, text and value input, menu generation, line printer output with page formatting, searching and sorting operations, and many others, we feel that the program development time for many of your business applications programs can be significantly reduced.

You will be able to concentrate on writing the true applications portion of the program instead of establishing a core program design and data structure.

The General Purpose Modules in the next section are designed to be useful not only at this point in your programming experience, but also as modules you can use in the future.

There are disadvantages in using the modules. It does take some time to learn how they work. They do establish a data structure that may not be the best one for many applications. However, give them a try. At the worst, you may gain some insights into system operation while we're explaining their functions. At best you'll have a standard set of general purpose programs that can form the basis for a "library" of other routines.

Section II
General Purpose Modules

Chapter Three
Overall Description of the General Purpose Modules

In this chapter we'll look at what the General Purpose Modules will do for you, how they are arranged in the program, what variables are used, how they work in general, and other aspects of using them. In later chapters in this section we'll look at common application program **functions** and how they are implemented by specific General Purpose Modules.

What the General Purpose Modules Are

The General Purpose Modules are a collection of about a dozen and a half modules, or collections of BASIC code. The module size varies from about four to fifty lines, in "uncompressed" format.

Each module performs a common function that is used again and again in every application program. There is a module to search a string for a given string (to find USA in SOUSA for example), a module to find a given entry in a string array, a module to print a report on the system line printer.

Figure 3-1 shows the MENU module as a typical module in size, complexity, and format.

```
10000 GOTO10140 'MENU
10010 '*************************************************************
10020 ' THIS IS THE MENU MODULE. IT TAKES A NUMBER OF STRING
10030 ' ITEMS AND DISPLAYS THEM ON THE SCREEN IN THE FORM OF A
10040 ' "MENU". IT THEN USES THE INPUT SUBROUTINE TO GET THE
10050 ' PONSE FOR VALIDITY. IF THE RESPONSE IS CORRECT, IT
10060 ' RETURNS THE NUMBER OF THE SELECTED ITEM. IF THE RES-
10070 ' PONSE IS NOT CORRECT, IT WAITS UNTIL A PROPER RESPONSE
10080 ' HAS BEEN KEYED IN BY THE USER.
10090 '      INPUT: ZA=# OF ITEMS 1-10
10100 '             ZA$(0)=TITLE OF MENU
10110 '             ZA$(1)-ZA$(N)=MENU SELECTIONS
10120 '     OUTPUT:ZB=ITEM SELECTED
10130 '*************************************************************
10140 IF ZA<1 OR ZA>10 THEN STOP
10150 CLS
10160 PRINT CHR$(2);
10170 ZI=LEN(ZA$(0))
10180 PRINT @YA/2-ZI/2,ZA$(0);
10190    FOR ZI=1 TO ZA
10200    PRINT@ZI*YA+YA+10,ZI;ZA$(ZI);
10210    NEXT ZI
10220 PRINT @ZI*YA+YC+15,"ENTER SELECTION, 1 THROUGH ";ZA;
10230 ZC=(ZI*YA+YC+50):ZD=2:ZE=0:GOSUB 2000
10240 IF ZF<1 OR ZF>ZA GOTO 10220
10250 ZB=ZF
10260 RETURN
```

Figure 3-1. MENU Module

Each module is a subroutine that is called by the "main" applications code — the part that you will normally be writing. The last command in every module is a RETURN to return back to the "calling" program. A typical "calling sequence" is shown in Figure 3-2, which calls the MENU module to write out a menu of items as shown in Figure 3-3.

```
20150 'DISPLAY MENU
20160 ZA=8:ZA$(0)="MAIL LIST":ZA$(1)="ADD ENTRY TO FILE"
20170 ZA$(2)="MODIFY OLD ENTRY":ZA$(3)="DELETE ENTRY"
20180 ZA$(4)="DISPLAY/PRINT FILE":ZA$(5)="SEARCH FILE"
20190 ZA$(6)="NEW SORT":ZA$(7)="LOAD FILE":ZA$(8)="SAVE FILE"
20200 GOSUB 10000
```

Figure 3-2. Calling Sequence

There are four parts to every module.

The first line has the name of the module. This is a descriptive name indicating what the module does, for example, "MENU" for the menu module.

The second part is a description of the module. In this case lines 10010-10130 are the description. When this module is used in your BASIC program, these lines would normally be deleted, since they would take up a great deal of storage and slow down program execution.

The third part of each module is a "body" of BASIC code. This is the portion of the module that actually does the computational work, for example, displaying the menu items on the screen. Within the body of the module, there may be one or more "GOSUBs" to other modules. In general, many of the modules are related in this fashion.

The fourth part of the module is the RETURN command.

When all of the descriptive parts of the module are deleted, the module appears in "compressed" form. The compressed form of the MENU module is shown in Figure 3-4. Note that all REMark lines have been left out, and that all spaces have been deleted, except for those within strings to be printed. This is the form of the module that you should use in your programs; the "uncompressed" form is for descriptive purposes only, and you'll have to admit, it is a lot easier to read. (An attempt to compress the program further by deleting every other statement failed, and is not included here . . .). The compressed form of all General Purpose Modules is provided in Appendix I.

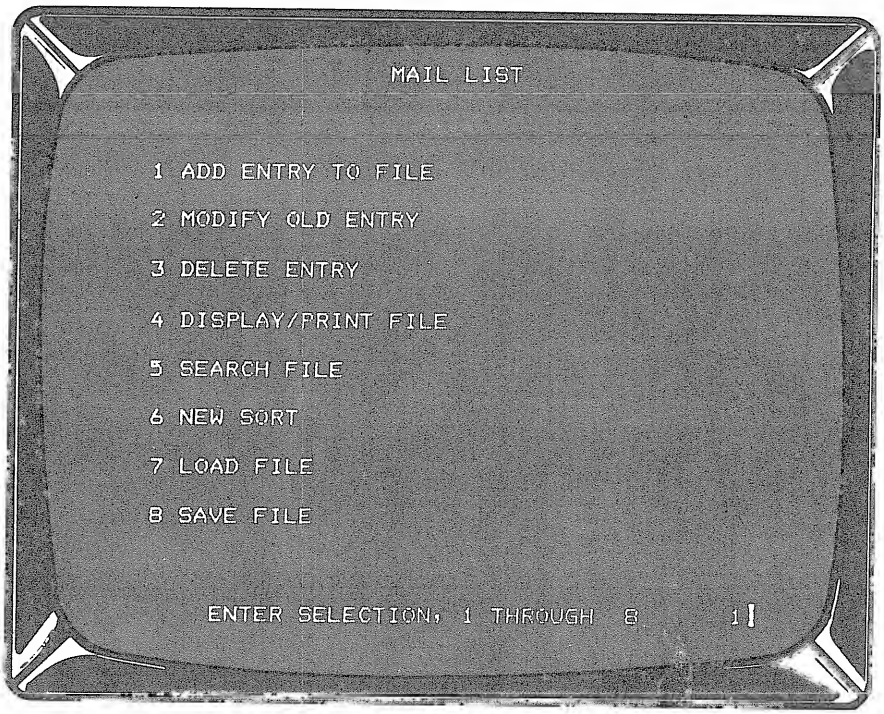


Figure 3-3. MENU Operation

```
10000 GOTO10140'MENU
10140 IFZA<1ORZA>10THENSTOP
10150 CLS
10160 PRINTCHR$(2);
10170 ZI=LEN(ZA$(0))
10180 PRINT@YA/2-ZI/2,ZA$(0);
10190 FORZI=1TOZA
10200 PRINT@ZI*YA+YA+10,ZI;ZA$(ZI);
10210 NEXTZI
10220 PRINT@ZI*YA+YC+15,"ENTER SELECTION, 1 THROUGH ";ZA;
10230 ZC=(ZI*YA+YC+50):ZD=2:ZE=0:GOSUB2000
10240 IFZF<1ORZF>ZAGOTO10220
10250 ZB=ZF
10260 RETURN
```

Figure 3-4. "Compressed" Form of MENU

How the Modules Are Arranged in a Program

A typical business applications program that uses the General Purpose Modules must have a definite line number arrangement, as shown in Figure 3-5. The General Purpose Modules have line numbers that range from line number 100 through line number 19999. The modules are arranged in this sequence for a specific reason.

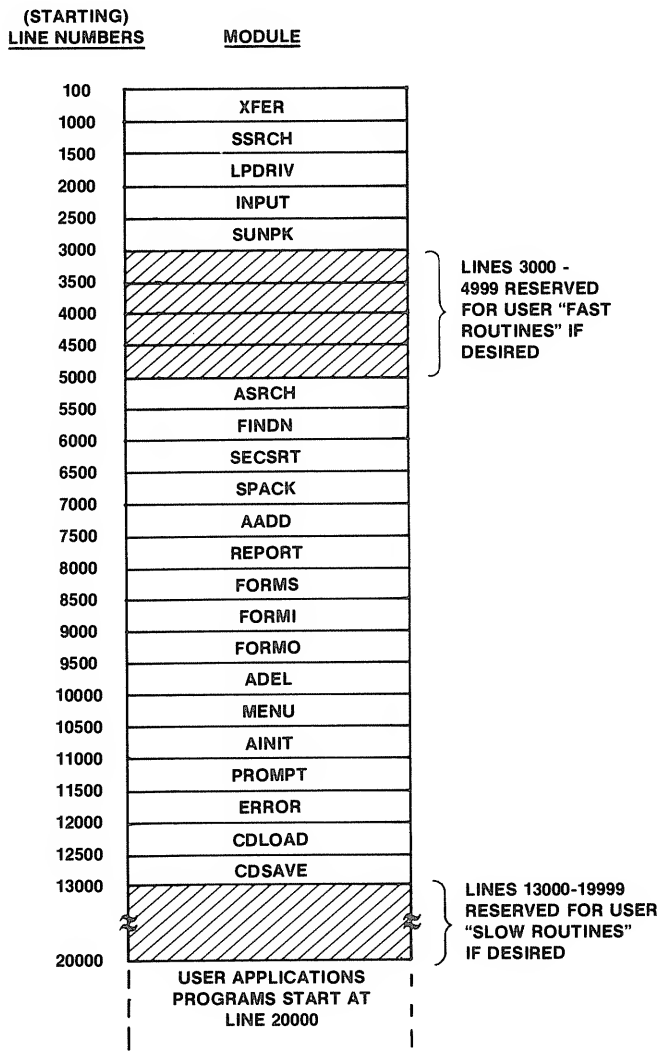


Figure 3-5. Line Numbers for General Purpose Modules and Other Programs

When the BASIC interpreter searches for a line number in a GOTO or GOSUB statement, it starts at the beginning line and searches forward. Even though this search is in **machine-language**, the language that the microprocessor in the TRS-80 understands (and few other parties), the search does take some time. This search time **can** make a difference in BASIC program execution speeds, hence we've put the most commonly used modules in the lowest line numbers.

Normally, when you are using the General Purpose Modules, you will be writing your BASIC applications code starting at line number 20000. Since your application code will do a lot of "calling" of the General Purpose Modules, the overall effect will be to speed up the entire applications program.

The General Purpose Modules take up about 5900 bytes of memory storage in compressed format. If you **must** delete any General Purpose Modules, there are some that cannot be deleted because other modules "call" them. Figure 3-6 shows how the modules interrelate and which ones could be deleted. You could not delete the INPUT module and still use MENU, for example, as there is a GOSUB to MENU in INPUT.

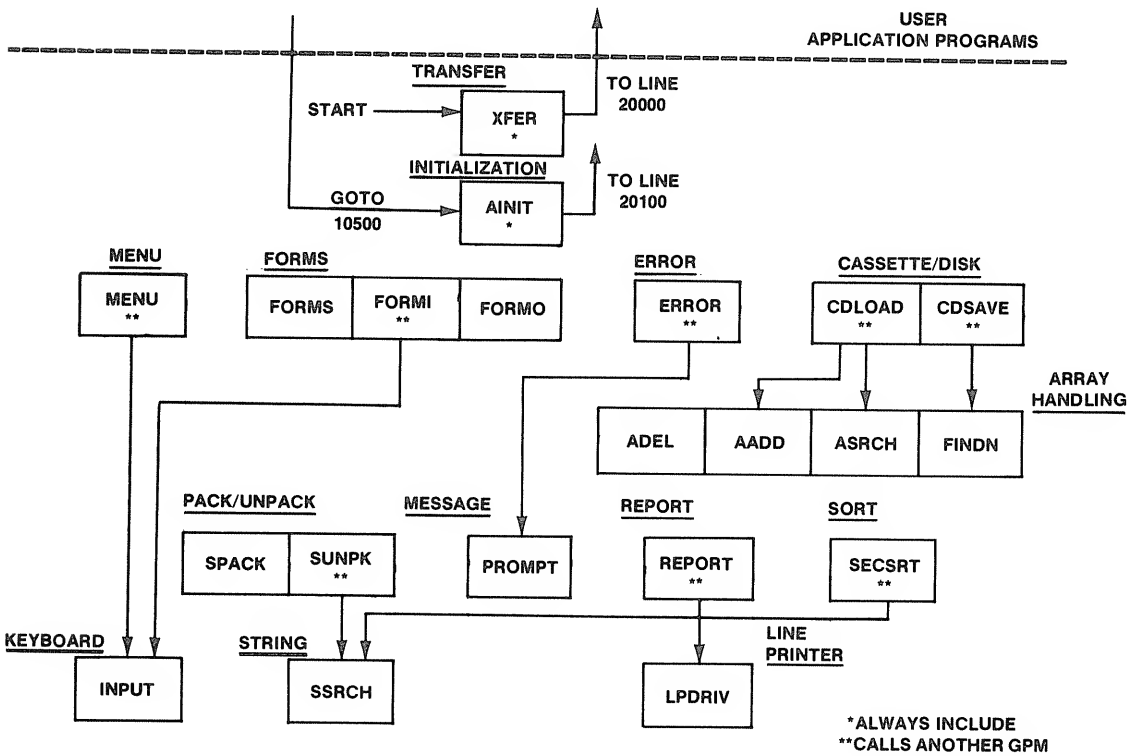


Figure 3-6. Relationship of General Purpose Modules

None of the modules are absolutely required for any applications program in the 20000 area except for the AINIT module. This module must be called **immediately** on start-up of your applications program. It must be called by a GOTO 10500 rather than a GOSUB, and line 10890 in AINIT must be changed to return back to the line after the GOTO 10500. We'll give some examples of this later in the book.

What Variables are Used

The General Purpose Modules use variables that start with the letters X, Y, and Z. These are "reserved" variable letter ranges and must not be used by your application program except as "inputs" or "outputs" to the modules. (My apologies to any Xaviers, Youngs, and Zieglers out there.) Any other variable names may be used by your application program, subject to the normal rules governing variable names.

The X, Y, and Z variables are defined as "integer" variables in the AINIT module. Integer variables, as you may know, take up only two bytes of storage and are rapidly accessed and handled by the BASIC interpreter.

The General Purpose Modules also use string variables in the same letter ranges. Many times a string variable may have the same two-letter name as an associated "integer" variable. This is permissable in BASIC. The string variable ZF$, for example, is different from the integer variable ZF, but both are used in the same module.

Arrays used by the General Purpose Modules have the same X,Y, and Z names. There is one large string array used, XA$, two large integer arrays, XA% and XB% (the "%" specifies "integer"), and several smaller arrays. The number of "elements" in each large array — XA$, XA%, and XB% — are equal and are "dimensioned" automatically based on available memory.

Figure 3-7 shows the variables used by the General Purpose Modules. Many of them are "shared" by several modules. There are two reasons for this. The fewer variables, the faster that each can be located by the BASIC interpreter, and hence, the faster the BASIC program will operate. Secondly, the modules are somewhat interrelated as a "system" and variables used in one module may provide information to the next module. We'll describe this action in the course of talking about the modules.

The shaded variables represent variables that you will never use; they are used as "scratch," or temporary storage, by the modules of the system.

VARIABLE	DESCRIPTION
XA%	ARRAY HOLDING PRIMARY POINTERS TO XA$.
XA$	MAIN ARRAY HOLDING DATA IN GPM.
XB%	ARRAY HOLDING SECONDARY POINTERS TO XA$.
XB	INPUT TO PROMPT FOR TYPE OF INPUT.
XB$	INPUT TO PROMPT FOR MESSAGE STRING.
XC	USER RESPONSE OUTPUT FROM PROMPT.
XC$	USER RESPONSE OUTPUT FROM PROMPT.
XD$	SEARCH STRING INPUT TO ASRCH.
XF	WORKING VARIABLE.
XH	WORKING VARIABLE.
XI	WORKING VARIABLE.
XI$	WORKING STRING.
XJ	OUTPUT FROM ASRCH. ENTRY BEFORE INSERTION POINT.
XK	OUTPUT FROM ASRCH. NEXT AVAILABLE ENTRY.
XL	OUTPUT FROM ASRCH. ENTRY AFTER INSERTION POINT.
XM	"FOUND" FLAG OUTPUT FROM ASRCH.
XN	REPORT COUNTER.
XP	ARRAY HOLDING ITEMS FOR REPORT.
XQ	SIZE OF XA%, XB%, XA$ ARRAYS.
XS	INPUT TO FINDN. # OF ENTRY TO FIND.
XT	INPUT TO FINDN. SEARCH TYPE FLAG.
XU	WORKING VARIABLE FOR FINDN.
XW	INDEX OUTPUT FROM SSRCH.
XW$	INPUT TO SSRCH. STRING TO BE FOUND.
XX	ABORT FLAG FROM INPUT.
XY$	INPUT TO SUNPK. STRING TO UNPACK.
XZ$	INPUT TO SSRCH. STRING TO BE SEARCHED.
YA	WIDTH OF SCREEN FROM AINIT.
YB	NUMBER OF LINES ON SCREEN FROM AINIT.
YC	WIDTH * 2 FROM AINIT.
YD	WIDTH * 3 FROM AINIT.
YE	WIDTH * 15 + 10 . PROMPT LOCATION FROM AINIT.
YF	BOTTOM BOX SYMBOL FROM AINIT.
YG	TOP BOX SYMBOL FROM AINIT.
YH	LEFT BOX SYMBOL FROM AINIT.
YI	RIGHT BOX SYMBOL FROM AINIT.
YJ	FIELD PROMPT CHARACTER FROM AINIT.
YK	CLEAR CHARACTER FROM AINIT.
YL	ACTIVITY FIELD START FROM AINIT.
YS	PRIMARY/SECONDARY FLAG.

"SYSTEM" PARAMETERS

Figure 3-7. General Purpose Module Variables

VARIABLE	DESCRIPTION
YT	SORT FIELD # INPUT TO SECSRT.
ZA$	ARRAY HOLDING MENU ITEM STRINGS.
ZA	INPUT TO MENU. NUMBER OF ITEMS.
ZB	OUTPUT FROM MENU. ITEM SELECTED.
ZC	SCREEN LOCATION INPUT TO INPUT.
ZD	MAXIMUM LENGTH INPUT TO INPUT.
ZE	NUMERIC/MIXED FLAG INPUT TO INPUT.
ZE$	WORKING STRING IN INPUT.
ZF	INPUT VALUE FROM INPUT.
ZF$	INPUT STRING FROM INPUT.
ZH	WORKING VARIABLE.
ZI	WORKING VARIABLE.
ZJ	LPDRIV FIRST TIME FLAG.
ZK	LINE COUNT IN LPDRIV.
ZL	LENGTH OF PAGE INPUT TO LPDRIV.
ZM	LENGTH OF PRINT IMAGE INPUT TO LPDRIV.
ZM$	STRING INPUT TO LPDRIV.
ZN$	TITLE STRING INPUT TO LPDRIV.
ZP	FORM WIDTH INPUT TO FORMS.
ZP$	ARRAY HOLDING ITEM STRINGS FOR FORMS.
ZQ	NUMBER OF ITEMS IN FORM INPUT TO FORMS.
ZR	ARRAY HOLDING ITEM LENGTHS FOR FORMS.
ZS	ARRAY HOLDING SCREEN LOCATIONS FROM FORMS.
ZW$	INPUT STRINGS FROM FORMI.
ZX	STRING LENGTHS FROM SUNPK.
ZZ	ARRAY HOLDING ERROR CODES FOR ERROR.
ZZ$	ARRAY HOLDING ERROR MESSAGES FOR ERROR.

Figure 3-7. General Purpose Module Variables

What the Modules Do

Each module accomplishes a specific function, sometimes without calling another module, and sometimes calling one or more. We'll describe each one starting from the most basic up. This will be the "first cut" explanation. Later we'll explain each module in detail.

XFER Module (Line 100)

This module simply transfers control to line 20000. Line 20000 is assumed to be the start of your application program for mail list, information retrieval, inventory, and so forth. When the entire application has been coded, specifying RUN without a line number will then start the program at line 20000. (The modules do get somewhat more complicated from this point . . .) Figure 3-8 shows the description of the module.

```
100 'XFER
110 '*************************************************************
120 GOTO 20000
130 '*************************************************************
```

Figure 3-8. XFER Module Description

SSRCH Module (Line 1000)

SSRCH is the search string module. Figure 3-9 shows its description. SSRCH looks for string XW$ in string XZ$. Suppose that you had defined a string in your applications program as A$ and you wanted to see if the "sub-string" BILL was in A$. The calling sequence to SSRCH might be

```
20200  INPUT A$
20210  XZ$=A$:XW$="BILL":GOSUB 1000
20220  REM RETURN HERE FROM SSRCH
```

String A$ was first input. Then string XZ$ was set equal to A$. Now there are two strings that contain the same character data, A$ and XZ$. XW$ was set equal to BILL. Next, a call was made to ASRCH by GOSUB 1000.

```
1000 GOTO 1090 'SSRCH
1010 '*************************************************************
1020 ' THIS IS THE SEARCH STRING MODULE. IT SEARCHES THE XZ$
1030 ' STRING FOR A SEARCH STRING OF XW$. XZ$ MUST BE LARGER
1040 ' OR EQUAL TO LENGTH OF XW$.
1050 '      INPUT: XW$=STRING TO BE FOUND
1060 '             XZ$=STRING TO BE SEARCHED
1070 '      OUTPUT:XW=START INDEX OF STRING OR -1 IF NOT FOUND
1080 '*************************************************************
```

Figure 3-9. SSRCH Module Description

When a RETURN is made to line 20220, variable XW will contain the location of BILL if found in XZ$ (A$), or a -1 if the BILL string was not found. If BARDEN,BILL was input as A$, for example, XW would be "returned" as 8, the eighth character position from the left in BARDEN,BILL.

Since we want to do these searches all the time, what you have in SSRCH is a handy "precanned" subroutine to search for a given string. Sure, it takes some shuffling around to set XW$ equal to A$, but that's the price you have to pay for precanned subroutines.

LPDRIV Module (Line 1500)

This module prints a string ZM$ on the system line printer. The description of LPDRIV is shown in Figure 3-10. If the string is a null (empty, as in ZM$=" "), then a "form feed" is done to the next page. This module automatically does "page formatting," skipping over the bottom of the page to start a new page. You can define the number of lines to be printed per page by setting variable ZM

to the number of lines per page and variable ZL to the length of the page in lines. ZM and ZL would only need to be set every time a new page format was needed (probably only once per program).

```
1500 GOTO 1630 'LPDRIV
1510 '*****************************************************
1520 ' THIS IS LINE PRINTER DRIVER. IT OUTPUTS A SINGLE LINE
1530 ' TO THE LINE PRINTER, ADDS ONE TO THE LINE COUNT, AND
1540 ' TESTS TO SEE IF LAST LINE ON PAGE HAS BEEN REACHED. IF
1550 ' LAST LINE HAS BEEN REACHED, A "FORM FEED" IS DONE.
1560 '      INPUT: ZM=NUMBER OF LINES PER PAGE
1570 '             ZL=LENGTH OF PAGE IN LINES
1580 '             ZM$=STRING TO BE PRINTED. IF NULL ("") THEN
1590 '                 "FORM FEED" IS DONE
1600 '             ZN$=PAGE TITLE MESSAGE OR "" IF NO TITLE
1610 '      OUTPUT:LINE IS PRINTED ON SYSTEM LINE PRINTER
1620 '*****************************************************
```

Figure 3-10. LPDRIV Module Description

A sample call to print the line THIS IS A LINE would look something like this

```
20200 ZM=50:ZL=66
.
.
.
20600 ZM$="THIS IS A LINE":GOSUB 1500
```

The dots indicate other code between the line that "initializes" the line counts and the line that calls LPDRIV.

INPUT Module (Line 2000)

This module is used to input a string of characters by using the INKEY$ function in BASIC. Figure 3-11 shows the description. As the characters are input, they are "echoed" to the screen at a screen location specified by variable ZC, which holds a screen location value of 0 through 1023 (Model I/III) or 0 through 1919 (Model II). Variable ZD holds the maximum length of the input string to be input. Variable ZE specifies whether a text string or numeric variable is to be input. On RETURN, string variable ZF$ holds the input string if a text string was to be input, or variable ZF holds the numeric value.

```
2000 GOTO2110 'INPUT
2010 '*****************************************************
2020 ' THIS IS THE INPUT MODULE. IT INPUTS A NUMERIC VALUE OR
2030 ' STRING FROM A GIVEN SCREEN LOCATION BY USING THE INKEY$
2040 ' FUNCTION.
2050 '      INPUT: ZC=SCREEN LOCATION, 0 THROUGH 1023
2060 '             ZD=MAXIMUM LENGTH OF INPUT STRING
2070 '             ZE=0 IF NUMERIC STRING, =1 IF MIXED
2080 '      OUTPUT:ZF=VALUE IF NUMERIC
2090 '             ZF$=STRING IF MIXED
2100 '*****************************************************
```

Figure 3-11. INPUT Module Description

This module is primarily used for "form fill in." It is called by module FORMI, the FORM Input module. This module calls INPUT with a "form field" location specified in variable ZC. INPUT handles the job of getting input characters from the keyboard, displaying them on the screen, and checking for valid characters.

The input function is terminated either when the ENTER key is pressed, or after the maximum number of characters has been entered.

A call to read an input string from screen location 544 (about the center of the screen on a Model I/III) with an expected length of no greater than 10 characters would look like this:

```
20200 ZC=544: ZD=10: ZE=1: GOSUB 2000
20210 REM RETURN HERE WITH STRING IN ZF$
```

SUNPK Module (Line 2500)

This module is used to "unpack" a large string which is contained in string variable XY$. The module description is shown in Figure 3-12. The unpacking operation separates the string into sub-strings, or fields. The General Purpose Modules work with fields of data for form fill in, searches, and other operations. The fields for a mail list entry might be

(Field 1)	BARDEN
(Field 2)	BILL
(Field 3)	HACK WRITER
(Field 4)	250 N.S. MEMORY LANE
(Field 5)	COMPUTER CITY
(Field 6)	CA
(Field 7)	99999
(Field 8)	(714) 555-1212

```
2500 GOTO 2600 'SUNPK
2510 '*****************************************************
2520 ' THIS IS THE UNPACK STRING MODULE. IT FINDS THE INDIV-
2530 ' IDUAL FIELDS OF A STRING CREATED BY THE PACK STRING
2540 ' MODULE BY LOOKING FOR AN EXCLAMATION MARK CHARACTER.
2550 '      INPUT: XY$=STRING TO UNPACK
2560 '      OUTPUT:STRING XY$ IS UNPACKED INTO ZW$(1)-ZW$(N)
2570 '             AND LENGTH OF EACH PUT INTO ZX(1)-ZX(N)
2580 '             NUMBER OF FIELDS IS PUT INTO ZQ.
2590 '*****************************************************
```

Figure 3-12. SUNPK Module Description

When the fields are stored as a complete entry in an array of mail list entries, they are stored like this

```
BARDEN!BILL!HACK WRITER!250 N.S.  MEMORY LANE!COMPUTER
CITY!CA!99999!(714) 555-1212!
```

The SUNPK module "unpacks" this entry and stores the eight fields by looking for the "!" character (called a delimiter) into the ZW$ array. The length of each of the fields is put into array ZX. The number of fields found is put into variable ZQ.

SUNPK is used to unpack any string into the number of fields defined by the delimiting "!" character.

ASRCH Module (Line 5000)

This module is used to search the XA$ array for a given string XD$. Figure 3-13 shows the description. The XD$ array holds a number of entries that have the "packed" form shown in the previous SUNPK description. These entries could represent mail list entries, billing information, orders, or any alphabetic, numeric, or special characters for the application involved.

```
5000 GOTO5140 'ASRCH
5010 '*************************************************************
5020 ' THIS IS THE SEARCH ARRAY MODULE. IT IS USED TO SEARCH
5030 ' FOR A GIVEN STRING, EITHER TO FIND THE EXPECTED STRING
5040 ' OR TO FIND THE SPOT WHERE THE STRING SHOULD BE INSERTED
5050 '      INPUT: XD$=STRING FOR SEARCH
5060 '             XA% AND XA$ ARRAYS CONTAIN APPROPRIATE DATA
5070 '      OUTPUT:XJ=INDEX TO ENTRY BEFORE STRING
5080 '             XK=INDEX TO NEXT AVAILABLE SLOT
5090 '             XL=INDEX TO NEXT ENTRY AFTER STRING OR STRING
5100 '                OR -1 IF NEXT ENTRY OUT OF ARRAY
5110 '             XM=0 IF NOT FOUND, 1 IF FOUND, 2=OUT OF MEMORY
5120 '             XS=# OF ENTRY
5130 '*************************************************************
```

Figure 3-13. ASRCH Module Description

To speed up searches, sorts, and merges, the XA% array holds values that represent the order of entries. The string to be found is put into XD$, and a call is made to ASRCH. The string will either be found or will not be found. If the string is found, variable XM=1, otherwise it is 0.

Three variables are used to hold the position of the string in either the "found" or "not found" case. If the search string is found, XL contains the number of the XA$ array that matches the entry; if the search string is not found, XL points to the **next entry** after the position in XA$ where the search string **should have been**. The XA$ array is essentially in alphabetical order. If XL=-1 no "intermediate position" was found.

Variable XJ points to the previous entry in XA$ if the search string is found, or to the entry before the position where the string should have been.

Variable XK points to the next "available," or unused, position in XA$. XA$ normally contains a large number of unused positions, either positions at the end because the array has never been filled up, or ones interspersed throughout XA$ because entries have been deleted.

Variable XS holds the number of the entry in XA$; this is either the number of the "found" entry, or the number of the position where the entry "should have been" (the insert point).

A call to ASRCH to search for the entry BARDEN!BILL!HACK WRITER!250 N.S. . . . in XA$ would go something like this

```
20200 XD$=''BARDEN'':GOSUB 5000
20210 IF XL <>-1 PRINT XA$(XL) ELSE PRINT ''NOT FOUND''
```

Note that here the search was made for the first "field" of BARDEN. XL would point to BARDEN!BILL! . . . even though the entire string did not match. Much more about that later.

FINDN Module (Line 5500)

This module finds a given entry number in the XA$ array. Figure 3-14 shows the description. Instead of searching for a string in XA$, FINDN simply counts forward until the *n*th entry is found. Variable XS contains the number of the entry to find.

```
5500 GOTO5650 'FINDN
5510 '************************************************************
5520 ' THIS IS THE "FIND NTH ENTRY" MODULE. IT SEARCHES THE
5530 ' XA% ARRAY TO FIND EITHER A GIVEN ENTRY #, OR TO FIND
5540 ' THE NEXT ENTRY.
5550 '      INPUT: XU=CURRENT # FROM PREVIOUS FIND NTH
5560 '             XS=# TO FIND, 1 TO N OR -1 IF FIND ALL
5570 '             XT=0 IF FIND NTH, 1 IF FIND NEXT
5580 '             YS=0 IF PRIMARY, 1 IF SECONDARY
5590 '      OUTPUT:XM=0 IF ENTRY NOT FOUND ON FIND NTH, 1 IF
5600 '                  FOUND
5610 '             XJ=INDEX TO LAST ENTRY
5620 '             XL=INDEX TO NTH ENTRY OR NEXT ENTRY
5630 '                OR -1 IF NOT FOUND
5640 '************************************************************
```

Figure 3-14. FINDN Module Description

On RETURN, variable XM is set to 1 if the *n*th entry is found or to 0 if it was not found. It will not be found if the number sought is greater than the number of entries in XA$. Variables XJ and XL are set as they were in ASRCH, to the previous position and to the next position in XA$, respectively. XL is set to -1 if the entry number sought is greater than the number of entries in XA$.

Normally FINDN is used to search for the *n*th entry and then, having found the *n*th entry, it is used to access the next entries one at a time. This is accomplished by setting variable XT, which specifies "find *n*" if XT=0, or "find next" if XT=1. Variable XU should be set to 0 on the first call to FINDN and is used by FINDN thereafter.

Normally FINDN searches through the XA$ array on the basis of "field 1" being in alphabetical order. Field 1 would be the last name of a mail list, for example, and the "leftmost" field in the XA$ entry. A "secondary key" of any field may be used, however. In this case, FINDN searches for the *n*th entry based on the order of one of the other fields. Variable YS is set to 0 as FINDN is to be used with field 1, or to 1 if FINDN is to be used with another field. The SECSRT module description explains this function further.

A sample call to find the 11th entry of XA$, and then to find the remaining entries in XA$ would look like this

```
20200 XU=0: XS=11: XT=0: YS=0: GOSUB 5500
20210 IF XM<>0 THEN PRINT XA$(XL) ELSE STOP
20220 XS=-1: XT=1: GOSUB 5500
20230 IF XL<>-1 THEN PRINT XA$(XL) ELSE STOP
20240 GOTO 20220
```

SECSRT Module (Line 6000)

This module is used to resort the XA$ array by a field other than field 1. Figure 3-15 shows the description of the module. Normally the order of XA$ is based on field 1, which is the leftmost field of each entry in XA$. The field 1 portion of

```
BARDEN!BILL!HACK WRITER!250 N.S.  MEMORY LANE . . .
```

for example, would be "BARDEN." The order for field 1 is maintained in integer array XA%.

```
6000 GOTO 6100 'SECSRT
6010 '*****************************************************
6020 ' THIS IS THE SECONDARY SORT MODULE. IT IS USED TO SORT
6030 ' THE XA$() ARRAY BY A SECONDARY FIELD. THE INDICES TO
6040 ' THE SORT ARE HELD IN THE SECONDARY INDEX ARRAY XB%()
6050 '      INPUT: XA$() AND XA%() ARRAYS
6060 '             YT=# OF SECONDARY FIELD
6070 '     OUTPUT:SORTED ARRAY OF INDICES IN XB%() ARRAY
6080 '*****************************************************
```

Figure 3-15. SECSRT Module Description

SECSRT allows a resort of the XA$ array based on some other field. Variable YT is used to input the field number. The "secondary" sort could be done on field 7, for example, if field 7 were the zip code in a mail list. Integer array XB% is used to hold the order for the new field, while integer array XA% maintains the "primary sort" order for field one.

To call SECSRT to sort on field 5, for example, we'd have:

```
20200 YT=5:GOSUB 6000
```

The idea here is to use the first field for high-speed shuffling of entries, but to give a slower-speed capability to sort on any entry field.

SPACK Module (Line 6500)

SPACK does the reverse of SUNPK. It "packs" a number of fields into one long string. Figure 3-16 shows the SPACK description. The fields are held in string array ZW$. Variable ZQ holds the number of fields to be packed. On RETURN, the fields from ZW$ are packed into string XY$, with "!" characters between each field.

```
6500 GOTO 6600 'SPACK
6510 '*****************************************************
6520 ' THIS IS THE PACK STRING MODULE. IT CONSTRUCTS A STRING
6530 ' OF XY$ MADE UP OF THE "ZP$" FIELDS IN SEQUENCE.
6540 ' THE "DELIMITER" BETWEEN FIELDS IN THE XY$ STRING IS AN
6550 ' EXCLAMATION MARK CHARACTER.
6560 '      INPUT: ZQ=# OF ZP$ FIELDS
6570 '             ZW$(1)-ZW$(N)=FIELDS
6580 '     OUTPUT:XY$ STRING MADE UP OF FIELDS
6590 '*****************************************************
```

Figure 3-16. SPACK Module Description

If we had the fields A LOAF OF BREAD, A JUG OF WINE, and THOU in ZW$(1), ZW$L(2), and ZW$(3), for example, this call to SPACK

```
20200 ZQ=3: GOSUB 6600
20210 PRINT XY$
```

would result in a printout of

```
A LOAF OF BREAD!A JUG OF WINE!THOU!
```

SPACK is used to unpack entries from XA$ and separate them into fields for easy printing, modification, or other operations.

AADD Module (Line 7000)

The AADD module is used to add an entry to the XA$ array. Figure 3-17 shows the description of AADD. A call to ASRCH must precede the call to AADD to set up variables XJ, XK, and XL.

```
7000 GOTO 7070 'AADD
7010 '*****************************************************
7020 ' THIS IS THE ADD ENTRY MODULE. IT ADDS AN ENTRY TO THE
7030 ' XA$ ARRAY AND SETS APPROPRIATE XA% POINTERS
7040 '      INPUT: XJ,XK,XL SETUP FROM SEARCH MODULE
7050 '     OUTPUT:ENTRY ADDED
7060 '*****************************************************
```

Figure 3-17. AADD Module Description

Obviously, it is convenient to be able to add, delete, and modify (delete and add) entries to the XA$ array, and AADD provides the add capability.

REPORT Module (Line 7500)

The REPORT module provides the capability of printing a form or report on the system line printer. Figure 3-18 shows the description. REPORT scans through integer array XP and prints based on the code found in each element of the array.

```
7500 GOTO 7640 'REPORT
7510 '*****************************************************
7520 ' THIS IS THE REPORT MODULE. IT PRINTS A REPORT ON THE
7530 ' SYSTEM LINE PRINTER AS DEFINED BY A LIST OF "ITEMS".
7540 '      INPUT: XP(0)=NUMBER OF ITEMS
7550 '             XP(1)-XP(N)=ITEMS
7560 '             XN=AUTO-INCREMENTING COUNTER
7570 '     OUTPUT:ITEMS DEFINE PRINTING OF FIELD DATA
7580 '      ITEMS: 0=LINE FEED
7590 '             1-N=FIELD N STRING IN ZW$(1)-ZW$(N)
7600 '             -M=TAB TO M
7610 '             -1=PAGE EJECT
7620 '             -2=PRINT REPORT COUNTER XN
7630 '*****************************************************
```

Figure 3-18. REPORT Module Description

Codes are provided for printing any field found in array ZW$, for a line feed (new line), for a tab to any character position along a line, for a new page, or for an item count. Although this doesn't appear to be much at first glance, it is possible to print virtually any form by defining the proper codes and fields. Possible report formats are described in detail in a later chapter.

FORMS Module (Line 8000)

This module prints out a "form" on the screen. Figure 3-19 describes the module. A "boxed" form of variable width, title, variable number of fields, and variable input field length is automatically generated by this module.

```
8000 GOTO8130 'FORMS
8010 '*****************************************************
8020 ' THIS IS FORM SKELETON MODULE. IT OUTPUTS A FORM MADE
8030 ' UP OF A NUMBER OF "FIELDS". IT IS USED IN CONJUNCTION
8040 ' WITH THE FORM INPUT MODULE TO INPUT FORM DATA.
8050 '      INPUT: ZP=FORM WIDTH IN # OF CHARACTERS, 10-60
8060 '             ZQ=# OF ITEMS, 1-12
8070 '             ZP$(0)=FORM TITLE
8080 '             ZP$(1)-ZP$(N)=FIELD STRING
8090 '             ZR(1)-ZR(N)=FIELD LENGTH
8100 '     OUTPUT:FORM IS OUTPUT ON SCREEN
8110 '             ZS(1)-ZS(N)=SCREEN LOCATION OF FIELDS
8120 '*****************************************************
```

Figure 3-19. FORMS Module Description

Variable ZP defines the form width. Variable ZQ is the number of fields in the form. The first entry of array ZP$, ZP$(0), holds the title of the form. The remaining entries of array ZP$ hold the field text. The corresponding entries of array ZR hold the "field width" for the input section of the form.

On RETURN, array ZS holds the screen positions for each of the fields for input.

To output a form for a "part number" description, for example, the following code might be used:

```
20200 ZP=50: ZQ=3: ZP$(0)="PART NUMBER DESCRIPTION"
20210 ZP$(1)="PART NUMBER ": ZP$(2)="DESCRIPTION "
20220 ZP$(3)="MANUFACTURER" : ZR(1)=10: ZR(2)=30
20230 ZR(3)=20: GOSUB 8000
```

and would result in the form shown in Figure 3-20.

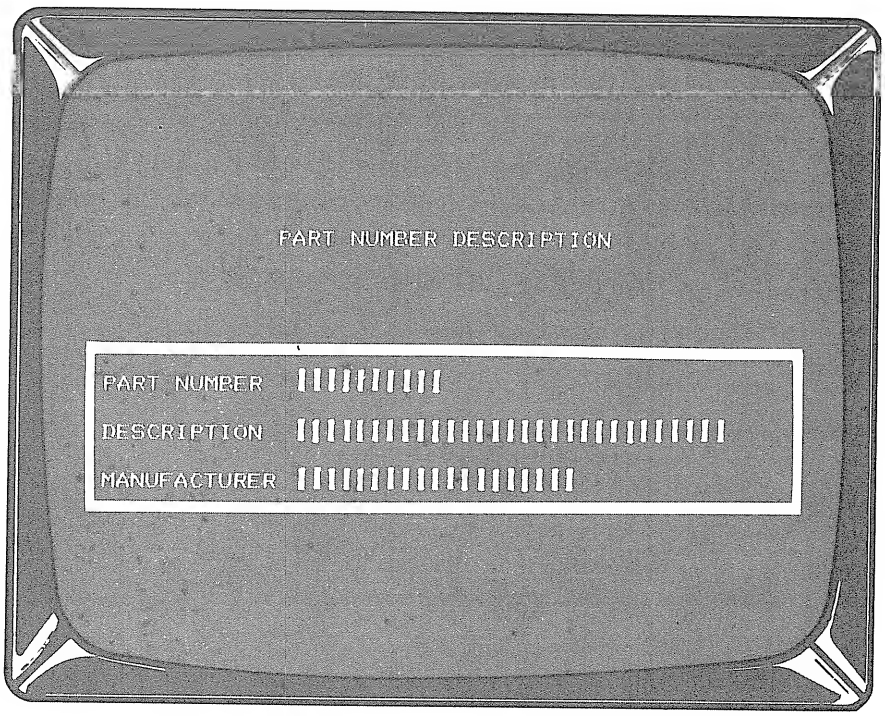


Figure 3-20. FORMS Module Example

ZS(1), ZS(2), and ZS(3) would hold three screen location values which could then be used in calling the FORMI module to input field data.

The FORMS module can have a great deal of use for displaying forms of various types since the forms only have to be defined one time, say in a subroutine. As any number of fields are allowed, and input field lengths are variable, the task of redefining new graphics and displays for different forms is handled automatically by FORMS.

FORMI Module (Line 8500)

This module is used after FORMS to automatically allow Input of field data on the displayed form. Figure 3-21 shows the module description. FORMI calls the INPUT module to input each form field. At RETURN, array ZW$ holds the entries for each field.

```
8500 GOTO8570 'FORMI
8510 '*****************************************************
8520 ' THIS IS THE FORM INPUT MODULE. AFTER A FORM HAS
8530 ' HAS BEEN OUTPUT, THIS MODULE READS IN THE FORM FIELDS.
8540 '      INPUT: NO PARAMETERS
8550 '     OUTPUT: INPUT STRINGS IN ZW$(1) - ZW$(N)
8560 '*****************************************************
```

Figure 3-21. FORMI Module Description

After FORMS has been called for the previous example of a part number description, FORMI would be called by

```
20240 GOSUB 8500
```

FORMO Module (line 9000)

This module is used in conjunction with FORMS to display field data. Figure 3-22 shows the description.

```
9000 GOTO9080 'FORMO
9010 '*****************************************************
9020 ' THIS IS FORM OUTPUT MODULE. IT MAY BE USED AFTER A FORM
9030 ' SKELETON HAS BEEN OUTPUT.  FIELD DATA MAY BE DISPLAYED
9040 ' BY USING THIS MODULE.
9050 '      INPUT: ZW$(1) - ZW$(N) CONTAINS FIELD DATA
9060 '     OUTPUT: FIELD DATA DISPLAYED IN FIELDS
9070 '*****************************************************
```

Figure 3-22. FORMO Module Description

FORMO is used to repetitively display new entries from XA$ (or any source) once FORMS has setup the "skeleton." Forty part number descriptions could be displayed in sequence by first calling FORMS, and by then calling FORMO forty times, after each part number entry was obtained from XA$ by FINDN and "unpacked" by SUNPK.

The field data to be displayed must be in array ZW$; this arrangement is performed automatically by SUNPK.

ADEL Module (Line 9500)

This module deletes an entry from XA$, based on variable setup from a previous ASRCH call. Figure 3-23 shows the description.

```
9500 GOTO9570 'ADEL
9510 '*****************************************************
9520 ' THIS IS THE DELETE ENTRY MODULE. IT DELETES AN ENTRY FROM
9530 ' THE XA$ ARRAY.
9540 '      INPUT: XJ,XK,XL SETUP FROM SEARCH MODULE
9550 '      OUTPUT: ENTRY DELETED
9560 '*****************************************************
```

Figure 3-23. ADEL Module Description

ADEL implements the "delete" portion of the add entry, modify entry, and delete entry to XA$.

MENU Module (Line 10000)

This module displays a menu of items and inputs an item number selection. Figure 3-24 shows the module description. Menus are used to provide interactive foolproof choice of system functions for applications programs. MENU outputs a menu of items with title, waits for the input defining the item number, and then returns the number selected.

```
10000 GOTO10140 'MENU
10010 '*****************************************************
10020 ' THIS IS THE MENU MODULE. IT TAKES A NUMBER OF STRING
10030 ' ITEMS AND DISPLAYS THEM ON THE SCREEN IN THE FORM OF A
10040 ' "MENU". IT THEN USES THE INPUT SUBROUTINE TO GET THE
10050 ' PONSE FOR VALIDITY. IF THE RESPONSE IS CORRECT, IT
10060 ' RETURNS THE NUMBER OF THE SELECTED ITEM. IF THE RES-
10070 ' PONSE IS NOT CORRECT, IT WAITS UNTIL A PROPER RESPONSE
10080 ' HAS BEEN KEYED IN BY THE USER.
10090 '      INPUT: ZA=# OF ITEMS 1-10
10100 '             ZA$(0)=TITLE OF MENU
10110 '             ZA$(1)-ZA$(N)=MENU SELECTIONS
10120 '      OUTPUT:ZB=ITEM SELECTED
10130 '*****************************************************
```

Figure 3-24. MENU Module Description

Variable ZA holds the number of items in the menu. The strings defining the items in the menu are in the ZA$ array, starting at ZA$(1). ZA$(0) holds the title of the menu. On RETURN, ZB holds the number of the item selected.

As an example, if you wished to display a menu of three choices for a billing system, the call might look like this:

```
20200 ZA=3: ZA$(0)=''ACCOUNT RETRIEVAL''
20210 ZA$(1)=''OLD ACCOUNT'': ZA$(2)=''NEW ACCOUNT''
20220 ZA$(3)=''RETURN'': GOSUB 10000
```

and the menu displayed would appear as shown in Figure 3-25. MENU would RETURN with a valid selection in variable ZB.

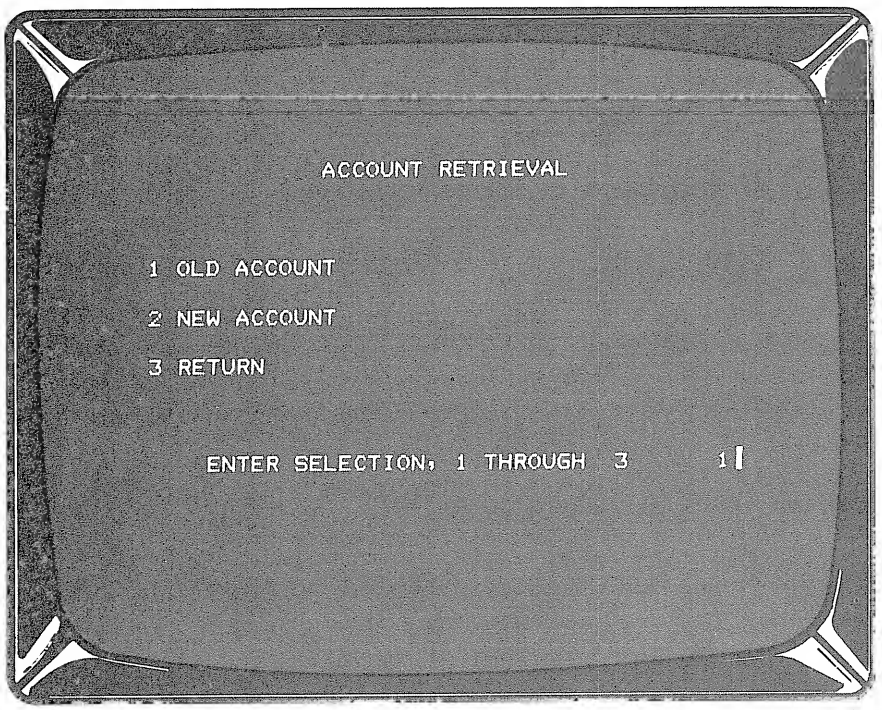


Figure 3-25. MENU Module Example

AINIT Module (Line 10500)

This module is used for initializing the arrays and some variables used by the General Purpose Modules. The description of AINIT is shown in Figure 3-26. This module should be called immediately by the applications program at 20000 by a GOTO 10500. Line 10890 of the module must be changed to the return line after the call.

```
10500 GOTO10610 'AINIT
10510 '*****************************************************
10520 ' THIS IS THE INITIALIZE ARRAYS MODULE. IT MUST BE CALLED
10530 ' BEFORE ANY ARRAY PROCESSING IS DONE. IT SETS UP THE
10540 ' XA% AND XA$ ARRAY TO "UNUSED".
10550 '      INPUT:  NO PARAMETERS
10560 '              ***CALLED BY A GOTO***
10570 '      OUTPUT: XA% ARRAY SET TO ALL -2
10580 '              XA$ ARRAY SET TO ALL "*"
10590 '*****************************************************
```

Figure 3-26. AINIT Module Description

The AINIT module calculates the amount of free memory and bases the size of the arrays upon this amount. It then initializes the XA$ and XA% arrays by marking each of their entries with an "unused" string or value. It then sets

parameters for the type of system by asking for the system type, Model I, II, or III.

PROMPT Module (Line 11000)

The PROMPT module is used to output a warning or prompt message on the last screen line. It then reads in a numeric or string response, or waits a short time if no response is expected. Figure 3-27 shows the description of this module.

```
11000 GOTO 11110 'PROMPT
11010 '*************************************************************
11020 ' THIS IS THE "PROMPT" MODULE. IT OUTPUTS A GIVEN MESSAGE
11030 ' AT LAST LINE AND READS IN A USER STRING OR NUMERIC
11040 ' RESPONSE.
11050 '      INPUT: XB$=MESSAGE TO BE OUTPUT
11060 '             XB=0 IF NUMERIC RESPONSE,=1 IF STRING,
11070 '               =2 IF YES OR NO RESPONSE,=3 NO RESPONSE
11080 '      OUTPUT:XC$=STRING RESPONSE OR "Y" OR "N"
11090 '             XC=NUMERIC RESPONSE OR 0 IF ENTER
11100 '*************************************************************
```

Figure 3-27. PROMPT Module Description

String variable XB$ is the message to the output. Variable XB is a 0 for a numeric response, 1 for a string response, 2 for a yes or no response, and 3 for no response. On RETURN string variable XC$ holds the string response or "Y" or "N," while integer variable XC holds a possible numeric response.

PROMPT is used for general system prompt messages and error messages. The following call outputs the error message INVALID ENTRY #:

```
20200 XB$=''INVALID ENTRY #'': XB 3 : GOSUB 11000
```

ERROR Module (Line 11500)

The ERROR module is used for "expected" errors, such as not finding a disk file name. The description of ERROR is shown in Figure 3-28. Certain types of errors are what we've melodramatically called "catastrophic" errors. These are "logic"

```
11500 GOTO 11640 'ERROR
11510 '*************************************************************
11520 ' THIS IS THE ERROR MODULE. IT RESPONDS TO A "NON-CAT-
11530 ' ASTROPHIC" SYSTEM ERROR BY OUTPUTTING A MESSAGE AND
11540 ' SETTING AN ERROR FLAG.
11550 '      INPUT: ZZ(0)=ZERO IF ALL ERRORS CATASTROPHIC,
11560 '                   NUMBER OF ERROR TYPES IF POSSIBLE
11570 '                   ERRORS
11580 '             ZZ ARRAY HOLDS ALL ERROR NUMBERS POSSIBLE
11590 '             ZZ$ ARRAY HOLDS ERROR MESSAGES
11600 '      OUTPUT:IF ERROR NUMBER FOUND, ERROR MESSAGE OUTPUT,
11610 '             XX FLAG SET, AND RETURN MADE TO NEXT STATE-
11620 '             MENT. IF NOT FOUND, RESUME AT ERROR.
11630 '*************************************************************
```

Figure 3-28. ERROR Module Description

errors caused by improper coding in your application; they need to be fixed by finding a bug or bugs in your program. ERROR passes these through with the message CATASTROPHIC ERROR, causing a BREAK.

If you allow certain errors in your program, such as not finding a named disk file, or attempting to write a protected diskette, ERROR gives you the ability to detect these errors, printout a message indicating the type of error, and continue in the program for corrective action. This is normally much better than a BREAK back to the BASIC interpreter for errors which are correctable by the system user.

Array ZZ is used to hold the "allowable" error numbers in ZZ(1) on. ZZ(0) holds the number of allowable error codes. A related array ZZ$ holds the error message to be displayed if an allowable error is found. After the detection of the error code and display of the message, variable XX is set so that the applications program knows that the ERROR routine has been entered.

CDLOAD Module (Line 12000)

This module is used to load a cassette or disk file in special General Purpose Module Format. The description of CDLOAD is shown in Figure 3-29.

```
12000 GOTO 12110 'CDLOAD
12010 '*****************************************************
12020 ' THIS IS THE CASSETTE/DISK LOAD MODULE. IT LOADS A
12030 ' CASSETTE OR DISK FILE IN GPM FORMAT INTO THE XA$ ARRAY
12040 ' AND ADJUSTS THE XA% POINTERS. THE LOAD IS EITHER AN
12050 ' INITIALIZE TYPE OR A MERGE.
12060 '      INPUT: ZW$(1)="C" FOR CASSETTE OR "D" FOR "DISK"
12070 '             ZW$(2)=FILENAME (DISK ONLY)
12080 '             ZW$(3)="I" FOR INITIALIZE OR "M" FOR MERGE
12090 '     OUTPUT:FILE READ INTO XA$ AND XA% ADJUSTED
12100 '*****************************************************
```

Figure 3-29. CDLOAD Module Description

General Purpose Module files are sequential "ASCII" files made up of all of the entries from the XA$ array. The CDLOAD routine reloads a previously written file into the XA$ array. There are two types of loads, initialize loads and "merges." The initialize load clears the XA$ array of all entries and loads in the cassette or disk entries from the specified file; it is similar to a BASIC LOAD. The merge load merges entries from the cassette or disk file with the current entries in the XA$ array, ordering the entries as it does so; it is similar to the BASIC MERGE.

CDLOAD is called with string array ZW$ containing the types of operations. ZW$(1) defines whether the load is from cassette (C) or disk (D). ZW$(2) is the disk file name. ZW$(3) defines an initialize or merge.

CDSAVE (Line 12500)

The description of CDSAVE is shown in Figure 3-30. CDSAVE is the counterpart to CDLOAD — it saves the data from XA$ to a disk or cassette file. The entries from

```
12500 GOTO 12580 'CDSAVE
12510 '*****************************************************
12520 ' THIS IS THE CASSETTE/DISK SAVE MODULE. IT SAVES THE XA$
12530 ' ARRAY AS A CASSETTE OR DISK FILE IN GPM FORMAT.
12540 '      INPUT: ZW$(1)="C" FOR CASSETTE OR "D" FOR DISK
12550 '             ZW$(2)=FILENAME (DISK ONLY)
12560 '     OUTPUT:XA$ ARRAY WRITTEN TO CASSETTE OR DISK
12570 '*****************************************************
```

Figure 3-30. CDSAVE Module Description

XA$ are written out as a file, from entry number 1 on through the last entry of XA$. The resulting GPM file must be reloaded by CDLOAD, but may be examined by the DOS LIST or PRINT command, as it is an ASCII file.

The ZW$ array holds the type of file and name as in the case of CDLOAD.

Unused Module Area

The line numbers from 13000 through 19999 have been left free for modules to be defined by the user. If each module is allocated a range of 500, there is enough room for 14 additional modules. If each module is numbered with lines in increments of 10, there may be 50 lines in a module.

Lines 3000 through 4999 have been left free for "high-speed" modules you might want to add to the General Purpose Modules. Modules located here will be executed more rapidly than in a later area.

Detailed Description of Modules

In the remainder of this section we'll describe each of the modules in detail. You may want to scan over this and go right on to Section III, which describes a sample application, and Section IV, which describes further applications. The module description, however, also discusses system operation in regard to display, keyboard input, string operations, array and data storage, and input/output, so you may want to refer back to it or read those portions of it which aren't detailed descriptions of the modules.

Chapter Four
Display Operations Using the GPM

We're going to discuss video display operations using the General Purpose Modules (GPM) in this chapter. We'll start by describing the display characteristics of the TRS-80, then describe what general approach is used in the GPM, and finally discuss three GPM display modules, MENU, FORMS, and PROMPT, in detail.

Video Display Characteristics

The screens of the TRS-80 Models I, II, and III are divided up into **lines** and **character positions**. The Model I and III have 16 lines, with each line capable of holding 64 characters in 64 character positions. The Model II has 24 lines with 80 characters on a line. The displays for the three models are shown in Figure 4-1.

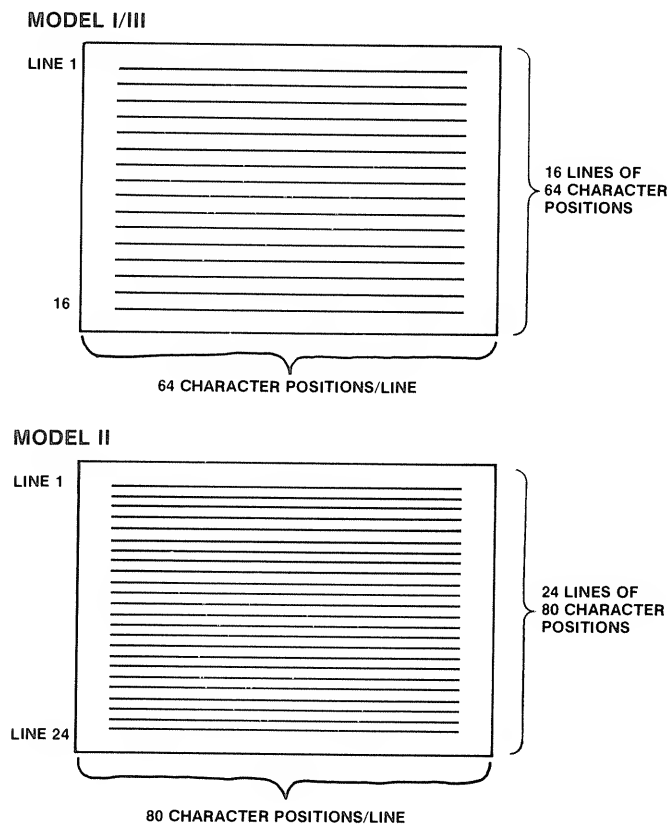


Figure 4-1. Video Display for Models I, II, and III

Most of the time we're interested in displaying **character data** on the screen. Whenever we talk about character data, or character **mode** we really mean display of the alphabetic characters A-Z and a-z, the digits 0-9, and special characters such as !, ", and &. These characters are represented by distinct codes called **ASCII** codes. ASCII (American Standard Code for Information Interchange) codes are standardized codes used by many input/output devices that convert "human" input to data, or that convert computer output to recognizable form. The ASCII codes used in the TRS-80s are shown in Figure 4-2.

SPECIAL CONTROL CODES			SPECIAL CHARACTERS		DIGITS		SPECIAL CHARACTERS		ALPHABETIC		SPECIAL CHARACTERS		ALPHABETIC		SPECIAL CHARACTERS	
0	16		32	␣	48	0	58	:	65	A	91		97	a	123	
1	17		33	!	49	1	59	;	66	B	92		98	b	124	
2	18		34	"	50	2	60	<	67	C	93		99	c	125	
3	19		35	#	51	3	61	=	68	D	94		100	d	126	
4	20		36	$	52	4	62	>	69	E	95		101	e	127	
5	21		37	%	53	5	63	?	70	F	96		102	f		
6	22		38	&	54	6	64	@	71	G			103	9		
7	23		39	'	55	7			72	H			104	h		
8	24		40	(	56	8			73	I			105	i		
9	25		41	)	57	9			74	J			106	j		
10	26		42	*					75	K			107	k		
11	27		43	+					76	L			108	l		
12	28		44	,					77	M			109	m		
13	29		45	—					78	N			110	n		
14	30		46	.					79	O			111	o		
15	31		47	/					80	P			112	p		
									81	Q			113	q		
									82	R			114	r		
									83	S			115	s		
									84	T			116	t		
									85	U			117	u		
									86	V			118	v		
									87	W			119	w		
									88	X			120	x		
									89	Y			121	y		
									90	Z			122	z		

Notes:

␣ = space

character may or may not be "printable" depending upon system and device. Code may or may not be standardized. See appropriate manual.

Figure 4-2. ASCII Codes

Note that all ASCII codes that define character data are between decimal 32 and decimal 127.

It's not only convenient to be able to display character data, however. Sometimes it's also nice to be able to draw bar graphs, horizontal and vertical lines, or

special characters, such as flashing cursors. The TRS-80s have this **graphics** capability to some extent, and we can easily draw lines or special symbols, or even make figures.

The TRS-80 Models I and III have 16 times 64 or 1024 character positions, while the Model II has 24 times 80 or 1920 character positions. **Each one** of these character positions may hold a single alphanumeric or graphics character. We can intermix alphanumeric and graphics characters at will by outputting the proper code to the screen character position. If the code is between 32 and 127, the character will be alphanumeric. If the code is a **graphics character code** a graphics character will result. The graphics character codes are not really part of the ASCII codes. There are special codes above 127 as shown in Figure 4-3.

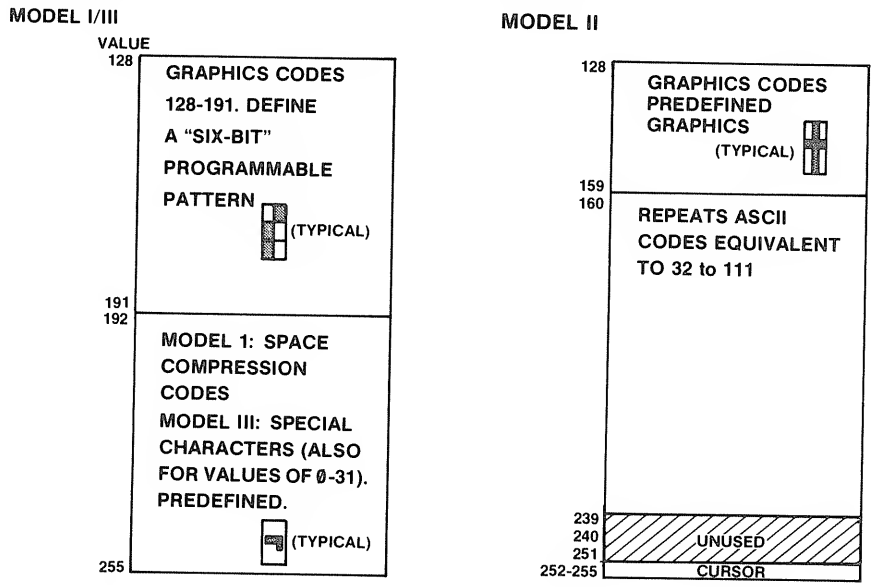


Figure 4-3. Graphics and Special Codes

Each of the character positions on the TRS-80 are referenced by a number, starting with 0. The first character position on line 1 is 0, the next is 1, and so forth. The first character position on subsequent lines is 64, 128, 196, and so forth for the Models I and III and 80, 160, 240, and so forth for the Model II. In other words, the character positions are incremented by 64 or 80 for each new line. To find the character position number for any character position, subtract 1 from the line number, multiply by 64 (Models I and III) or 80 (Model II), and add the character position on the line minus one.

Come on now, the math isn't all that bad. As an example, suppose that we wanted the middle of the screen on a Model I. There are 16 lines, so we'll take

line 8. There are 64 character positions per line, so we'll take the 32nd. The character position number (CP#) is

CP#=(8-1)*64+32-1=512+31=543

This scheme is shown in Figure 4-4. See, you didn't need a computer to work it out . . .

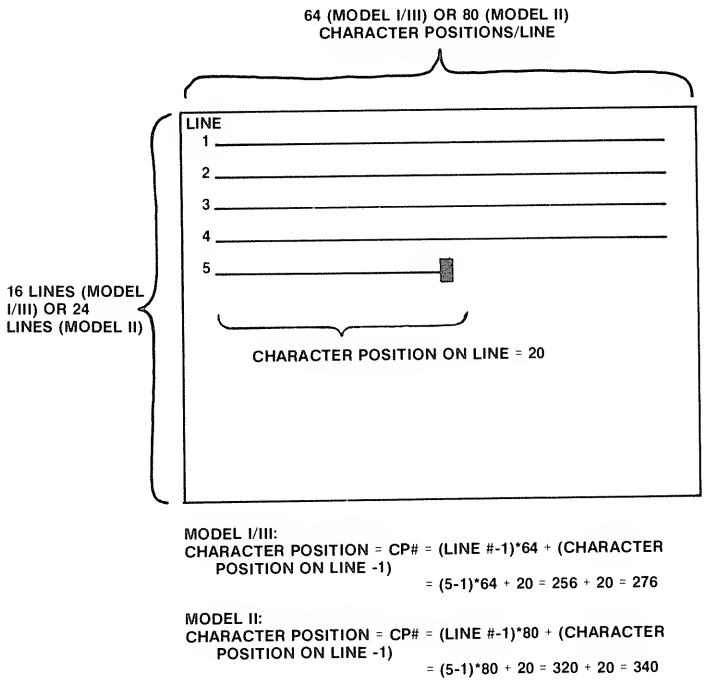


Figure 4-4. Character Position Numbering

BASIC Methods of Displaying Characters

There are really two methods for displaying alphanumeric or graphics characters on the video screen in BASIC, the PRINT or the PRINT @.

We can simply say

```
100 PRINT "HELP, I AM BEING HELD PRISONER IN A FORT WORTH COMPUTER PLANT"
```

and get an output of the text at the next line on the screen. The problem with this method is that the screen keeps "scrolling up"; subsequent lines eventually reach the bottom of the screen and previous lines disappear out of sight past the top. Also, we can't easily display a form, as the scrolling moves the form up.

The problems related to scrolling are solved by using the PRINT @ form of the PRINT command. Every time a PRINT @ is used, the character position of the

screen is specified after the @, and no scrolling occurs, as long as a semicolon (;) is used at the end of the print statement. The semicolon simply says, "print the text and don't skip to the next line."

Most of the display work we'll be doing in the GPM will use the PRINT @ command, rather than just a PRINT, so you'd better brush up on it in your reference manual if you're not familiar with it. The general form of the PRINT @ is:

```
100 PRINT @ XXXX,''TIPPECANOE AND TRS-80 TOO!'';
```

The XXXX will be a character position, 0 through 1023 for the Models I and III or 0 through 1919 for the Model II. The semicolon will always be used at the end to avoid skipping to the next line and disrupting previous screen output.

How about graphics character display? We'll be using graphics characters for display of forms. We'll use the graphics characters to draw a box around the forms, a simple application. We'll also use graphics characters as a blinking cursor for input of text, and to represent predefined "fields" of data. These functions are shown in Figure 4-5.

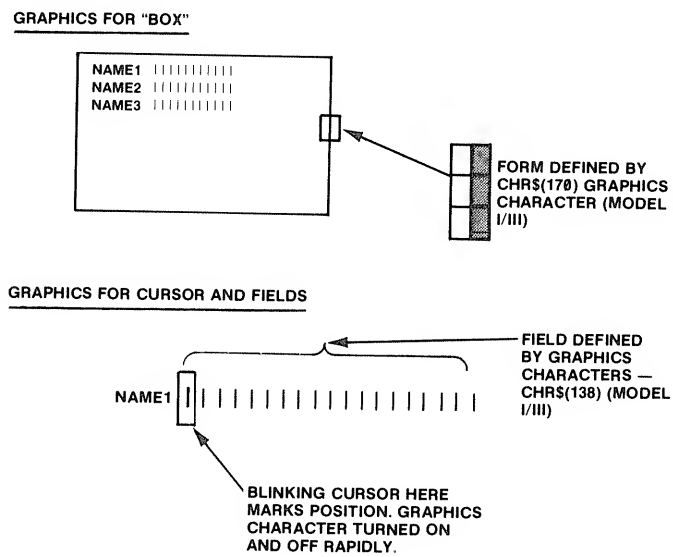


Figure 4-5. Graphics Display Operations

Sometimes we'll be PRINTing a string of graphics characters (as in the "box" lines), and other times we'll be PRINTing a single graphics character (as in the cursor case). The common method for doing this is to use the CHR$ function.

The CHR$ function creates a single character string from a numeric value. If we said

```
100 PRINT @ 512, ''MESSAGE1''+CHR$(149)+''MESSAGE2'';
```

for example, we'd be printing the string shown in Figure 4-6. The "+" in this case means that we've joined the three strings of MESSAGE1, CHR$(149), and MESSAGE2 to make one larger string, and then PRINTed that string. This process of joining strings is called **concatenation**, a bit of computer jargon for a simple process.

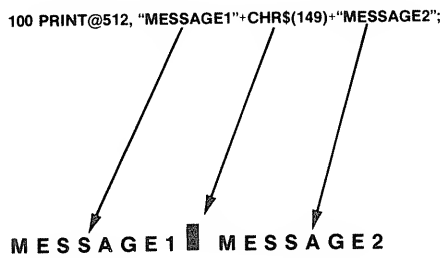


Figure 4-6. CHR$ Example

The CHR$ function created a single character string from the graphics character code of 149. This single character code was joined to MESSAGE1 and then MESSAGE2 was joined to the result.

If we wanted to output a number of graphics characters at one time, we could make a long string out of graphics characters by saying something like

```
100 A$=CHR$(149)+CHR$(149)+CHR$(149)
200 PRINT @ 512, A$;
```

which would print three graphics characters at screen character position 512.

An easier way to do this, provided that all graphics characters are the same, would be to use the STRING$ function. The STRING$ function creates a string made up of a number of characters, alphanumeric or graphics. To make a string of 5 "A" characters, for example, we'd say

```
100 B$=STRING$(5, ''A'')
200 PRINT @ 512,B$;
```

The result of this would be a display of ''AAAAA'' at screen character position 512.

To create a string of graphics codes, it's simply a matter of combining the STRING$ and CHR$ functions. This code

```
100 B$=STRING$(5, CHR$(149) )
200 PRINT @ 512, B$;
```

would print 5 graphics characters at screen position 512. This is the technique we'll use in the GPM for printing the horizontal lines of boxes, or for field characters.

GPM Design Philosophy for Display Operations

The basic idea we've used for the General Purpose Modules display functions is this: We've provided modules for easy display of menus, forms, and **system messages.**

We've made these displays automatic as far as screen formatting. You don't have to be concerned about screen character positions — for example, the module will automatically center the form for you. Menu items will be automatically numbered. System messages, that is, **messages** from your applications program, will be displayed in a special message area. An "activity area" will display counts of internal "number crunching" so that during long processes the user will be able to see that the system is actually working instead of being "hung up." (Yes, Virginia, systems even "hang up" for computer book writers — too often!)

The display modules work in conjunction with keyboard input modules. We'll explain the display functions in this chapter and talk about keyboard input in the next.

The screen has been divided into specific areas in the GPM design, as shown in Figure 4-7. The first line is generally used as a title for a form or menu. The last

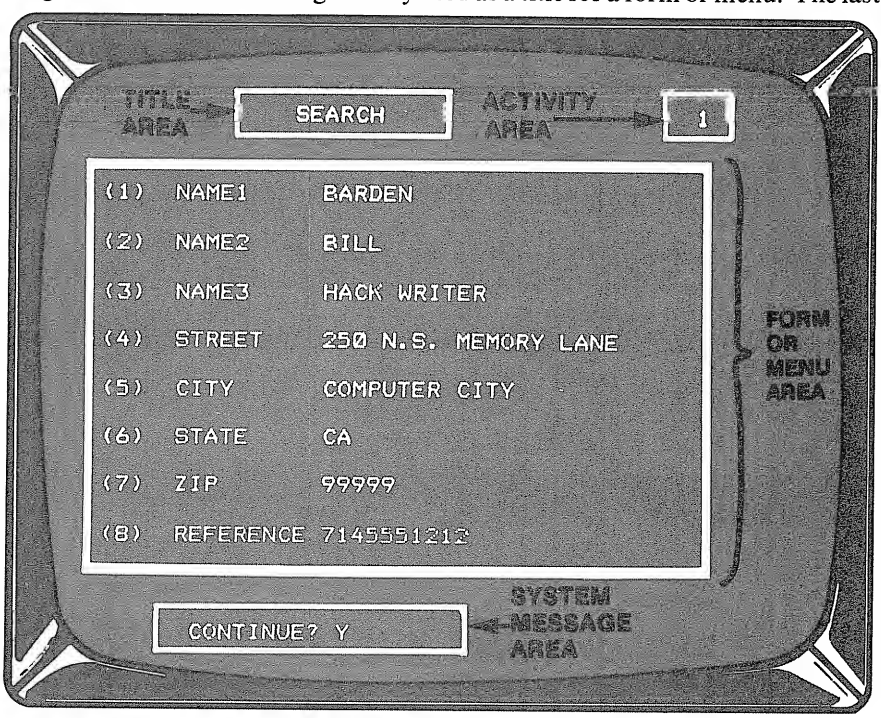


Figure 4-7. GPM Screen Areas

portion of the title line is the "activity area." It displays activity counts during extended processing. The activity counts are generally the **record numbers** of data being processed.

The last line of the screen has been designated as a system message area. The system message may be an error message — indication of an invalid record number, for example, or it may be a "prompt" type of message — a query regarding continuing or restarting.

The bulk of the screen is used to display a menu or form. Typical menus and forms are shown in Figure 4-8.

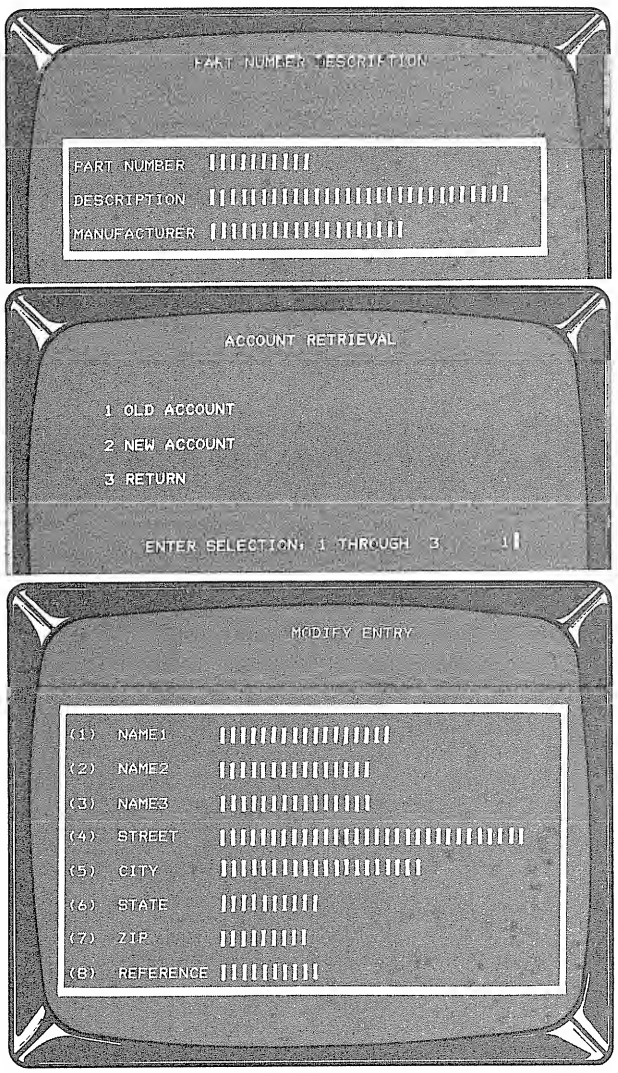


Figure 4-8. Menus and Forms

The system variables used in the display modules are automatically set after loading and specification of your system type (Model I, II, or III). Variable YA is the screen width in characters (64 or 80). Variable YB is the number of lines (16 or 24). Variables YC and YD are the character positions of the start of the second and third lines of the screen. Variable YF, YG, YH, and YI are the graphics characters to be used for horizontal "top" lines, horizontal "bottom" lines, "left" lines, and "right" lines. Variable YJ is the graphics character to be used for the input "field" area to be displayed on the form. Variable YE defines the character position of the input area for a response to a system message. Variable YL defines the character position of the activity area. All of these variables are initialized in the AINIT module.

MENU Module Operation

The MENU module is shown in Figure 4-9. It displays a number of menu items on the screen, along with a menu title, and calls the INPUT Module (line 2000) to input a menu item selection. The menu items and title are automatically centered and numbered. A typical call and display are shown in Figure 4-10.

```
10000 GOTO10140 'MENU
10010 '*****************************************************
10020 ' THIS IS THE MENU MODULE. IT TAKES A NUMBER OF STRING
10030 ' ITEMS AND DISPLAYS THEM ON THE SCREEN IN THE FORM OF A
10040 ' "MENU". IT THEN USES THE INPUT SUBROUTINE TO GET THE
10050 ' PONSE FOR VALIDITY. IF THE RESPONSE IS CORRECT, IT
10060 ' RETURNS THE NUMBER OF THE SELECTED ITEM. IF THE RES-
10070 ' PONSE IS NOT CORRECT, IT WAITS UNTIL A PROPER RESPONSE
10080 ' HAS BEEN KEYED IN BY THE USER.
10090 '      INPUT: ZA=# OF ITEMS 1-10
10100 '             ZA$(0)=TITLE OF MENU
10110 '             ZA$(1)-ZA$(N)=MENU SELECTIONS
10120 '      OUTPUT:ZB=ITEM SELECTED
10130 '*****************************************************
10140 IF ZA<1 OR ZA>10 THEN STOP
10150 CLS
10160 PRINT CHR$(2);
10170 ZI=LEN(ZA$(0))
10180 PRINT @YA/2-ZI/2,ZA$(0);
10190     FOR ZI=1 TO ZA
10200     PRINT@ZI*YA+YA+10,ZI;ZA$(ZI);
10210     NEXT ZI
10220 PRINT @ZI*YA+YC+15,"ENTER SELECTION, 1 THROUGH ";ZA;
10230 ZC=(ZI*YA+YC+50):ZD=2:ZE=0:GOSUB 2000
10240 IF ZF<1 OR ZF>ZA GOTO 10220
10250 ZB=ZF
10260 RETURN
```

Figure 4-9. MENU Module Listing

First, MENU tests the number of menu items, variable ZA. If ZA is less than 1 or greater than 10, MENU stops so that the application program can be corrected. Next, the screen is cleared by the CLS.

CALL:

```
20150 'DISPLAY MENU
20160 ZA=8:ZA$(0)="MAIL LIST":ZA$(1)="ADD ENTRY TO FILE"
20170 ZA$(2)="MODIFY OLD ENTRY":ZA$(3)="DELETE ENTRY"
20180 ZA$(4)="DISPLAY/PRINT FILE":ZA$(5)="SEARCH FILE"
20190 ZA$(6)="NEW SORT":ZA$(7)="LOAD FILE":ZA$(8)="SAVE FILE"
20200 GOSUB 10000
```

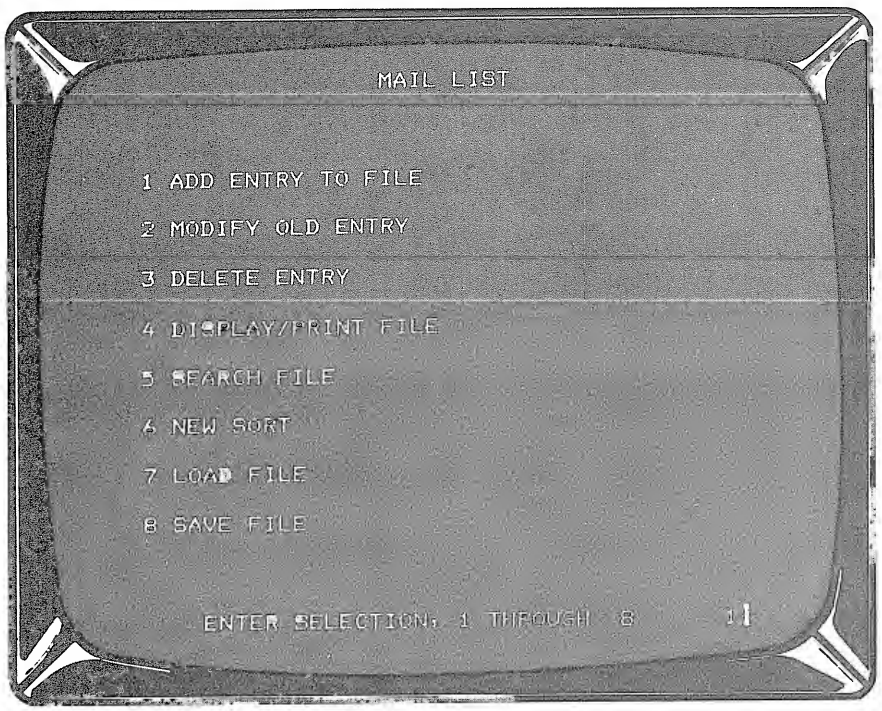


Figure 4-10. Menu Module Call and Display

Now, the title is displayed on the first line. The title is in ZA$(0) and may be any number of characters, as long as it does not overlap the "activity area," which starts 9 character positions less than the screen width. The length of the title is found by the LEN function, which is used to set ZI equal to the title length. The title in ZA$(0) is then printed at (YA-ZI)/2, which is width-length of title divided by two, or the character position which approximately centers the title. We say approximately, because the result could be XX.5. In this case, the "XX" position is automatically used by the PRINT @ function.

The menu selections are displayed in the "indented loop." The loop goes from ZI=1 to ZA, the number of items. Nine items, for example, would mean 9 repeats, or iterations. The PRINT @ statement in the loop prints the selection number (ZI) and the text of the selection (ZA$(ZI)). The "@" location is (ZI*YA+YA+10). This is equal to the width of a line times the iteration number plus one line plus 10 character positions. For the first item, this would be

1*64+64+10 or 138, ten character positions on the third line (Mod I, III width used in this example). Subsequent items are arranged in a column one line below.

Now the prompt message ENTER SELECTION, 1 THROUGH ZA is displayed two lines after the last item and slightly indented (ZI*YA+YC+15). YC is equal to two line widths (YA*2).

At this point a call is made to the INPUT subroutine to input the response. The parameters passed to INPUT (Line 2000) are the screen location for the input, ZC, the maximum length of the input string, ZD, and a "numeric/string" flag, ZE. Here the maximum length is two characters (values of 1 - 10), the input is numeric, and the location is 50 character positions past the prompt message.

A RETURN is made from INPUT with ZF set to the input value. A check is made on this value. If it is less than 1 or greater than the number of menu items (ZA), it is invalid, and the prompt message must be repeated for new input. If the value is all right, ZB is set equal to ZF and a return is made to the user's application program.

FORMS Module Operation

The FORMS module is shown in Figure 4-11. It displays a form on the screen, along with a form title. Inside the form are "fields," which are subdivisions of the

```
8000 GOTO8130 'FORMS
8010 '************************************************************
8020 ' THIS IS FORM SKELETON MODULE. IT OUTPUTS A FORM MADE
8030 ' UP OF A NUMBER OF "FIELDS". IT IS USED IN CONJUNCTION
8040 ' WITH THE FORM INPUT MODULE TO INPUT FORM DATA.
8050 '      INPUT: ZP=FORM WIDTH IN # OF CHARACTERS, 10-60
8060 '             ZQ=# OF ITEMS, 1-12
8070 '             ZP$(0)=FORM TITLE
8080 '             ZP$(1)-ZP$(N)=FIELD STRING
8090 '             ZR(1)-ZR(N)=FIELD LENGTH
8100 '      OUTPUT:FORM IS OUTPUT ON SCREEN
8110 '             ZS(1)-ZS(N)=SCREEN LOCATION OF FIELDS
8120 '************************************************************
8130 IF ZP<10 OR ZP>YA-4 THEN STOP
8140 IF ZQ<1 OR ZQ>12 THEN STOP
8150 CLS
8160 PRINT CHR$(2);
8170 PRINT @YA/2-LEN(ZP$(0))/2,ZP$(0);
8180 ZI=(YA-ZP)/2
8190 PRINT @ ZI+YC,STRING$(ZP,CHR$(YF));
8200 PRINT @ZI+YD+YA*ZQ,STRING$(ZP,CHR$(YG));
8210    FOR ZH=ZI+YD TO ZI+YD+YA*(ZQ-1) STEP YA
8220    PRINT @ ZH,CHR$(YH);
8230    PRINT @ ZH+ZP-1,CHR$(YI);
8240    NEXT ZH
8250    FOR ZH=1 TO ZQ
8260    PRINT @ZI+1+YC+YA*ZH,ZP$(ZH);" ";STRING$(ZR(ZH),CHR$(YJ));
8270    ZS(ZH)=ZI+1+YC+YA*ZH+LEN(ZP$(ZH))+1
8280    NEXT ZH
8290 RETURN
```

Figure 4-11. FORMS Module Listing

form. Each field has text defining it, and an entry area for user input. The text and size of the entry area are defined by the user. The form is "boxed" and automatically centered. A typical call and form are shown in Figure 4-12.

CALL:

```
29050 ZP$(0)="MODIFY ENTRY"
29060 ZP=57:ZQ=8
29070 ZP$(1)="(1)  NAME1     ":ZP$(2)="(2)  NAME2     "
29080 ZP$(3)="(3)  NAME3     ":ZP$(4)="(4)  STREET    "
29090 ZP$(5)="(5)  CITY      ":ZP$(6)="(6)  STATE     "
29100 ZP$(7)="(7)  ZIP       ":ZP$(8)="(8)  REFERENCE"
29110 ZR(1)=17:ZR(2)=15:ZR(3)=15:ZR(4)=30:ZR(5)=20
29120 ZR(6)=10:ZR(7)=9:ZR(8)=10:GOSUB 8000
```

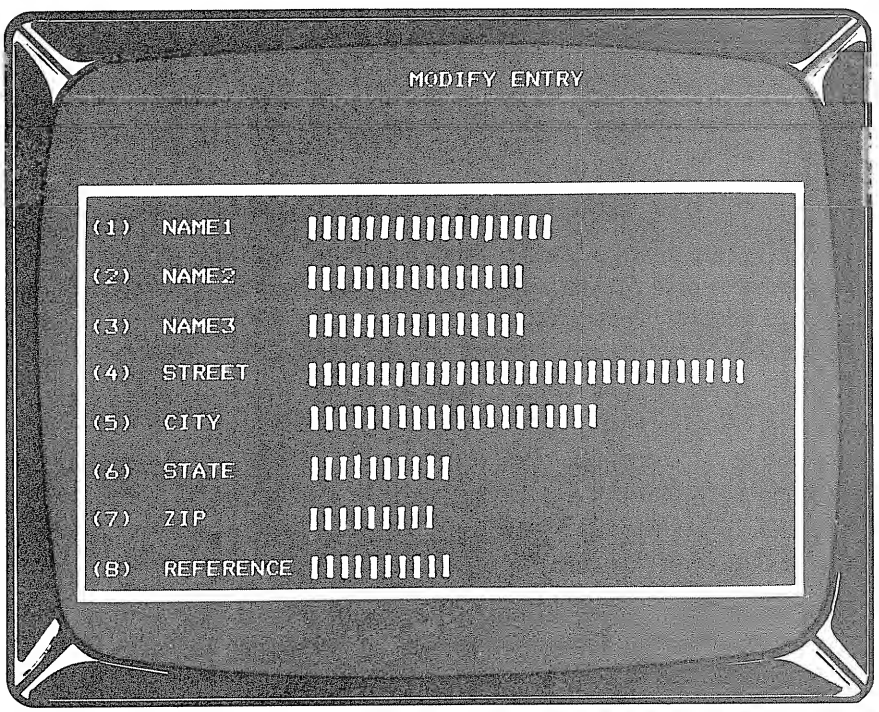


Figure 4-12. FORMS Module Call and Display

First the form width in ZP is compared to limits of 10 and 4 less than the screen width, YA. If the form is too small or too large, FORMS stops so that the application program can be corrected. Next, the number of fields in the form, ZQ, is tested. If ZQ is less than 1 or greater than 12 a second STOP occurs. If these two parameters are all right, the screen is cleared in preparation for displaying the form.

The title of the form is in ZP$(0). It is displayed at (YA-LEN(ZP$(0))/2. This is the width of the screen — the length of the title divided by 2, which gives the starting character position of the title for centering.

Variable ZI is set to (YA-ZP)/2. This is the left-hand edge of the form to properly center it.

The next two statements draw the top and bottom lines of the box. STRING$(ZP,CHR$(YF)) defines a string of graphics characters with a length equal to the form width ZP.

Variable YF is the proper graphics character for the system, established in AINIT. The string is PRINTED @ a position two lines (YC) down from top and "in" an amount equal to ZI.

STRING$(ZP,CHR$(YG)) defines a string of graphics characters of form width; it is drawn at ZI+YD+YA*ZQ. This is at the next line after the last of the field lines.

The next set of code (lines 8210 through 8280) represents a loop to draw the left and right lines for the box. ZI+YD is the character position one line down from the first character position of the top line. ZI+YD+YA*(ZQ-1) is the character position one line up from the first character position of the bottom line. ZH increments from the start to end character positions by STEPping YA. YA is the screen width or line width. STEPping this value increments to the next vertical character position. The left line is drawn by outputting the left graphics character (YH) at ZH. The right line is drawn by outputting YI at ZH+ZP-1.

At this point the box has been drawn. All that remains is to display the text for each field and the entry area for each field. ZH is again incremented, this time from 1 to ZQ, the number of fields. Each time through the loop, one field description, a blank, and a field entry area are displayed on the next line.

The field description is obtained from ZP$(ZH). A blank separates the field description from the field entry area. The field entry area is generated by STRING$(ZR(ZH),CHR$(YJ)). The YJ character is the graphics character for the field entry area from the AINIT module. The length of the field entry area is obtained from the ZR array. Each field, then, has text (from ZP$) and an entry area length (from ZR) associated with it as an input parameter.

The field description is printed at ZI+1+YC+YA*ZH, which starts at one character position further right than the left side, and at the current line for the field (YC+YA*ZH).

A second action taken in this loop is to save the starting character position of the field entry area in array ZS. This starting position is used in FORMI, form input, and FORMO, form output to define where the field entry areas are. Each field entry area is defined by ZI+1+YC+YA*ZH+LEN(ZP$(ZH))+1, which is the start of the text for the field plus the length of the text plus one.

A RETURN is made from FORMS with one entry in the ZS array for each field entry area (ZQ fields).

FORMO Module Operation

At this point, FORMS has displayed the form as defined by the calling applications program. There has been no text input or output in the field entry areas, and the form appears as shown in Figure 4-12. FORMO (Figure 4-13) is used to output to the field entry area after FORMS has been called. A typical call and output are shown in Figure 4-14.

```
9000 GOTO9080 'FORMO
9010 '*****************************************************
9020 ' THIS IS FORM OUTPUT MODULE. IT MAY BE USED AFTER A FORM
9030 ' SKELETON HAS BEEN OUTPUT.  FIELD DATA MAY BE DISPLAYED
9040 ' BY USING THIS MODULE.
9050 '      INPUT: ZW$(1) - ZW$(N) CONTAINS FIELD DATA
9060 '      OUTPUT: FIELD DATA DISPLAYED IN FIELDS
9070 '*****************************************************
9080    FOR ZI=1 TO ZQ
9090    PRINT @ ZS(ZI),ZW$(ZI);STRING$(ZR(ZI)-LEN(ZW$(ZI)),"  ");
9100    NEXT ZI
9110 RETURN
```

Figure 4-13. FORMO Module Listing

CALL:

```
21800 'DISPLAY MAIL LIST SKELETON
21810 GOSUB 29000
21820 'DISPLAY FIELDS ON FORM
21830 GOSUB  9000
```

ACTUAL CALL TO FORMO

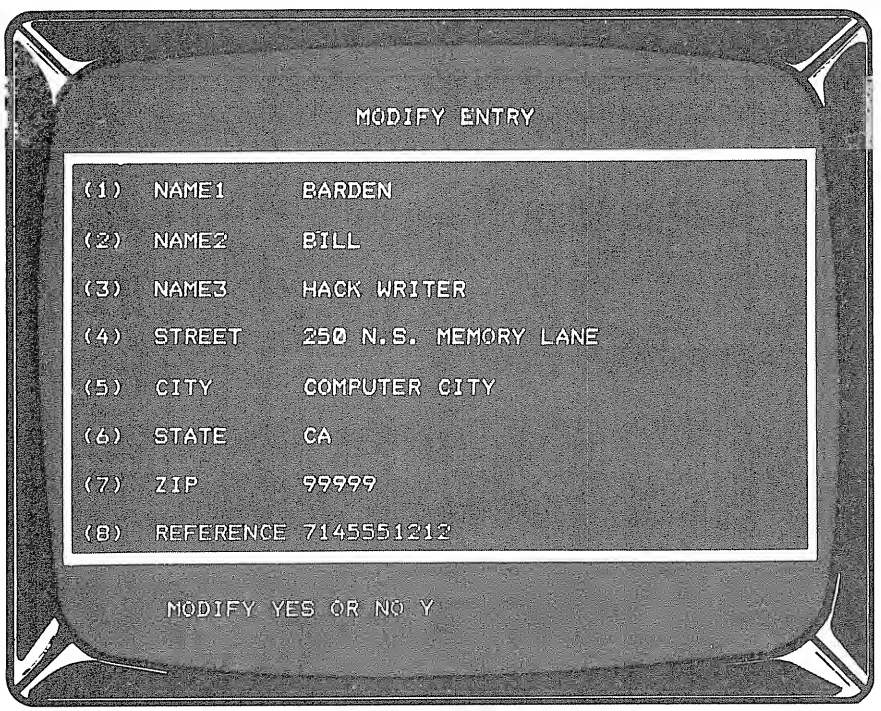


Figure 4-14. FORMO Module Call and Display

Text to be output to the field entry areas has to be in array ZW$ at this point. ZW$(1) corresponds to field 1, ZW$(2) to field 2, and so forth. FORMO is called with the ZW$ array containing the text to be output. FORMO then goes through the ZW$ array for ZQ field items. ZQ must be set to the number of fields associated with the form.

For each iteration of the ZI loop, the field entry area character position is picked up from the ZS array. ZS was previously set up by FORMS and contains the starting character position for every field entry area. A PRINT @ is done at ZS(ZI). The text from ZW$(ZI) is PRINTed. As FORMO is typically used to print records of a file in form format, any remaining graphics characters or previous characters in the field entry area are cleared. The length of the field entry area is in the ZR array. We also know the length of the text for the field — it is LEN(ZW$(ZI)). The difference of the two are the number of blank spaces required to substitute for previous characters. The required string is STRING$(ZR(ZI)-LEN(ZW$(ZI)),'' '').

A RETURN is made after the last field has been displayed in the form.

PROMPT Module Operation

The last module in this section is the PROMPT module, shown in Figure 4-15. It is used for four functions

```
11000 GOTO 11110 'PROMPT
11010 '*****************************************************
11020 ' THIS IS THE "PROMPT" MODULE. IT OUTPUTS A GIVEN MESSAGE
11030 ' AT LAST LINE AND READS IN A USER STRING OR NUMERIC
11040 ' RESPONSE.
11050 '      INPUT: XB$=MESSAGE TO BE OUTPUT
11060 '             XB=0 IF NUMERIC RESPONSE,=1 IF STRING,
11070 '               =2 IF YES OR NO RESPONSE,=3 NO RESPONSE
11080 '      OUTPUT:XC$=STRING RESPONSE OR "Y" OR "N"
11090 '             XC=NUMERIC RESPONSE OR 0 IF ENTER
11100 '*****************************************************
11110 XX=0
11120 PRINT @ YE,XB$+"                    ";
11130 IF XB=3 GOTO 11280
11140 XC$=""
11150    XI$=INKEY$:IF XI$="" GOTO 11150
11160    IF XI$>CHR$(YK) GOTO 11200
11170    IF XI$<>CHR$(YK) GOTO 11190
11180    XX=1: GOTO 11290
11190    IF XI$=CHR$(13) GOTO 11230
11200    XC$=XC$+XI$
11210    PRINT @ YE+LEN(XB$)+1,XC$;
11220    GOTO 11150
11230 IF XB=0 THEN XC=VAL(XC$)
11240 IF XB<>2 GOTO 11290
11250 IF XC$<>"YES" AND XC$<>"Y" AND XC$<>"NO" AND XC$<>"N" GOTO 11120
11260 XC$=LEFT$(XC$,1)
11270 GOTO 11290
11280    FOR XI=1 TO 900:NEXT XI
11290 RETURN
```

Figure 4-15. PROMPT Module Listing

- Displaying an error or warning message
- Displaying a message that requires a text answer
- Displaying a message that requires a YES or NO answer
- Displaying a message that requires a numeric answer

All four functions involve a message in XB$. This message is output in the "prompt" message area at character position YE. YE is defined in the AINIT module and is dependent on the system type. One of the first actions taken in PROMPT is to output the XB$ message at YE. If variable XB then specifies that "no response" is required (error or warning message), a GOTO timing loop FOR XI=1 TO 900:NEXT I is done. This loop simply wastes time for about 3 seconds while the TRS-80 user can read the message and then RETURNs to the calling program.

A typical call and display are shown in Figure 4-16.

CALL:

```
?1100 XB$="MODIFY YES OR NO":XB=1:GOSUB 11000:IF XX=1 GOTO ?1240
```

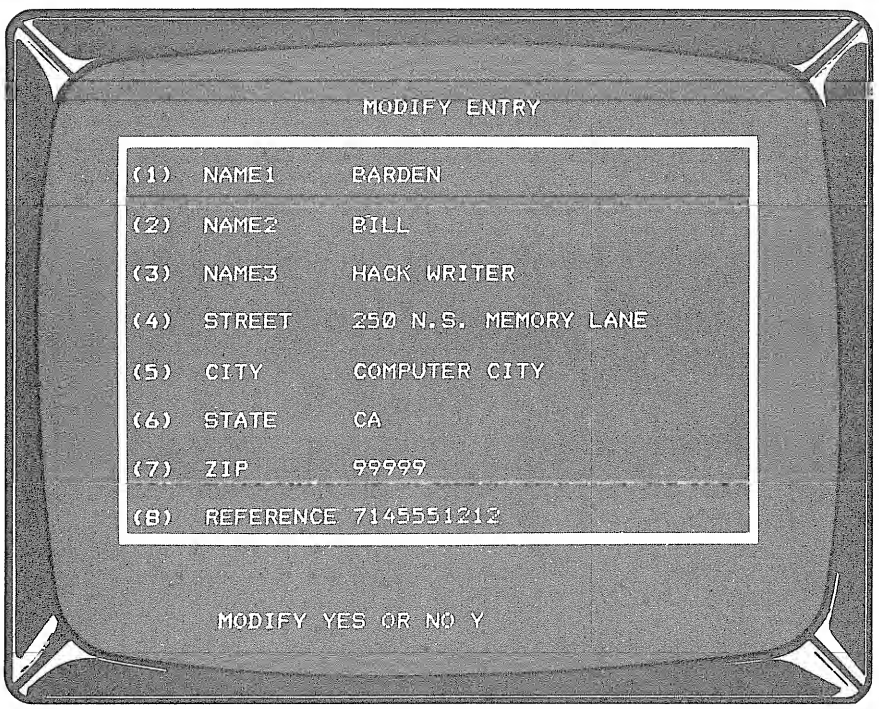


Figure 4-16. PROMPT Module Call and Display

Since the other three functions are very much related to user response and keyboard input, we'll discuss them in the next chapter.

Chapter Five
Character Input Using the GPM

Input of character data involves some special tricks using the INKEY$ function. We'll find out how input can be done in general, the design philosophy of the GPM, and how individual modules in the GPM work in this chapter.

Keyboard Input Operations

Whenever a key is pressed on the TRS-80 system, a single character code is generated. If the key pressed is an alphabetic character, a digit, or a special character such as " or #, the character code is in standard ASCII code, as we described in the last chapter.

There are certain keys, however, that generate codes that are not ASCII codes, but which are unique to the TRS-80 system. The right arrow, left arrow, and BREAK are examples of these types of **control** keys. In order to use the TRS-80 effectively, we've got to be able to read characters one at a time, using the INKEY$ function. The INKEY$ function will read all keys - even the control keys. Because of this we've got to delve into the mysteries of the "control codes" to a certain extent. (I know - somewhere in a padded room there is a programmer inventing computer jargon . . .)

Why can't we simply use the INPUT command of BASIC to input strings of character data? The reason is that the INPUT is done on a line basis. After the INPUT, the BASIC interpreter skips to the next line. If the line is at the bottom of the screen, it also **scrolls** the screen up one line, with the display going off the top edge of the screen. This automatic feature of BASIC is **too** automatic for us. We have no control over **where** the character string appears on the screen during input. This makes it virtually impossible to fill-in forms and to respond to messages at a specific screen location.

The INKEY$ Function

You may never have used the INKEY$ function in BASIC before, so let's review how it works. The basic code for an INKEY$ operation is shown here

```
100 CLS
200 A$=INKEY$
300 IF A$="" "" GOTO 200
400 PRINT @ 0, A$;
500 GOTO 200
```

This code first clears the screen by the CLS. Next, string A$ is set equal to INKEY$. When the BASIC interpreter encounters line 200, it will immediately read the keyboard. If no key is being pressed at the time, INKEY$ will be set equal to a **null string,** or "". If a key is being pressed, INKEY$ will be set equal to the character **or code** for that key.

What we've done in lines 200 and 300 is to test INKEY$ for a pressed key. If a key is being pressed, INKEY$ and A$ will be set equal to the key. If no key is being pressed, INKEY$ and A$ will be set to a null string of "". We'll loop back to line 200 until we detect a key.

When we detect a key we'll terminate the loop and print the one character string A$ at screen location 0, the start of the first line. Then we'll go back for the next character in the INKEY$ loop.

Here's a very important point: The INKEY$ function reads **but does not display** the key. We have to display the key ourselves by a PRINT statement. This is the price we have to pay for being able to detect a single key at a time without intervention by the "display driver" of the BASIC interpreter.

If you run the program above, you'll notice that the special control keys such as the arrows do not generally display anything on the screen. The reason for this is that the special **control codes** generated for these keys are not displayable when we go to PRINT them — the BASIC interpreter doesn't know what to do with them since they are not standard display characters.

We can see what codes are generated by a similar type of program to the one above. Line 400 here has an ASC function that converts the one-character string in A$ to a numeric value. This numeric value is an ASCII value for displayable characters or a control code value for non-displayable characters.

```
100 CLS
200 A$=INKEY$
300 IF A$="" "" GOTO 200
400 PRINT @ 0, ASC(A$);
500 GOTO 200
```

If we run this program, we can see the ASCII codes (check the back of your BASIC reference manual) **and** the control codes that are read from the keyboard. We've listed **all** the codes from a Model I using Disk BASIC in Figure 5-1. Other models or configurations may differ slightly.

What Do You Do With Those Control Codes?

The main problems in using INKEY$ are in PRINTing the characters at the right place and discarding useless control codes. However, we may keep **some** control codes for special functions.

One of the control codes we might want to keep is that generated for the BACKSPACE (Model II) or Left Arrow (Model I/III). This key is normally used to delete a character previously typed.

Another key that might be useful is the ENTER key. This key would mark the end of the input just as it does in the BASIC INPUT statement.

!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
=	61
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
0	48
:	58
—	45

S↑	(27)
S↓	(26)
S@	96
S←	(24)
S→	(25)
SCL	(31)
↑	91
↓	(10)
@	64
←	(9)
→	(8)
CL	(31)
SA	97
SB	98
SC	99
SD	100
SE	101
SF	102
SG	103
SH	104
SI	105
SJ	106
SK	107

SL	108
SM	109
SN	110
SO	111
SP	112
SQ	113
SR	114
SS	115
ST	116
SU	117
SV	118
SW	119
SX	120
SY	121
SZ	122
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72

I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90

<	60
>	62
+	43
?	63
,	44
.	46
/	47
;	59

S = SHIFT
(31) non-standard

Figure 5-1. Keyboard Codes

A third control key that might be useful is one that would say "stop doing whatever we are doing and get me back to some control point." This is somewhat similar to the BREAK key, which allows a return to the BASIC interpreter from any point in a BASIC program. A key that might be used here is the CLEAR (Model I/III) or "up arrow" (Model II).

Of course, these choices for usable control keys are somewhat arbritrary (I have a well-used dart board marked off in ASCII codes . . .). Any other keys that produce specific control codes could have been used instead. Depending upon the application, we might have also chosen keys that produce ASCII characters for control functions. If we never use the % character, for example, we could have used that key to designate "backspace."

We have in fact, set aside the ! character as a special character in the GPM modules. The exclamation point is used between fields of data, and can't be used in normal text input. (The GPM modules can't be used for advertising copy! If you know what I mean!)

GPM Design Philosophy for Input Operations

In the General Purpose Modules, we've provided modules for inputting character or numeric data by means of the INKEY$ function. At the same time that character data is entered, it is displayed by the GPM module on the screen.

Three control keys are used in the GPM:

1. Left arrow or BACKSPACE for backspace
2. ENTER for ending the input
3. CLEAR or "up arrow" for setting a "terminate operation" flag

The GPM character input modules work in conjunction with the GPM display modules discussed in chapter four. There are three GPM modules in this function, INPUT, FORMI, and PROMPT.

INPUT is the "lowest level" module. It allows entering character data via the INKEY$ function and automatically "echoes" the characters at a given screen location.

FORMI is the "FORM Input" module. It works in conjunction with FORMO, "FORM Output" to allow the user to enter data for each form field. FORMI calls the INPUT module to actually do the INKEY$ input.

The PROMPT module is a dual function module that both displays a system message and accepts a response to it. PROMPT was partially discussed in chapter four. PROMPT uses a separate INKEY$ "routine" from INPUT.

There are several system variables used with the character input modules. Variable YK is the "terminate operation" code. This is initialized in the AINIT module based upon whether the system is a Model I/III or Model II. A 31 code is used for the latter, while a 30 code is used for the Model II.

Variable YJ is the "field prompt" character, a vertical bar in the Model I/III or an asterisk in the Model II. This character is also initialized in AINIT based on the system. The field prompt character is automatically displayed whenever an INPUT operation is done so that the input field can be defined.

INPUT Module Operation

The INPUT module is shown in Figure 5-2. It inputs a string of characters and echoes the string to a given screen location. It is primarily used after the FORMO module (see Chapter Four) has output a form to the screen. A sample call and action are shown in Figure 5-3.

```
2000 GOTO2110 'INPUT
2010 '*****************************************************
2020 ' THIS IS THE INPUT MODULE. IT INPUTS A NUMERIC VALUE OR
2030 ' STRING FROM A GIVEN SCREEN LOCATION BY USING THE INKEY$
2040 ' FUNCTION.
2050 '      INPUT: ZC=SCREEN LOCATION, 0 THROUGH 1023
2060 '             ZD=MAXIMUM LENGTH OF INPUT STRING
2070 '             ZE=0 IF NUMERIC STRING, =1 IF MIXED
2080 '      OUTPUT:ZF=VALUE IF NUMERIC
2090 '             ZF$=STRING IF MIXED
2100 '*****************************************************
2110 XX=0
2120 IF ZC<0 OR ZC>YA*YB-1 THEN STOP
2130 IF ZD<1 OR ZD>255 THEN STOP
2140 PRINT@ZC,STRING$(ZD,CHR$(YJ));
2150 ZF$=""
2160       PRINT @ ZC+LEN(ZF$),CHR$(YJ);
2170       ZE$=INKEY$
2180       IF ZE$<>"" GOTO 2210
2190       PRINT @ ZC+LEN(ZF$)," ";
2200       GOTO 2160
2210    IF ZE$>CHR$(YK) GOTO 2300
2220    IF ZE$=CHR$(13) GOTO 2350
2230    IF ZE$<>CHR$(YK) GOTO 2250
2240    XX=1:GOTO 2360
2250    IF ZE$<>CHR$(8) GOTO 2160
2260    IF ZF$="" GOTO 2290
2270    ZF$=LEFT$(ZF$,LEN(ZF$)-1)
2280    PRINT @ ZC, ZF$+CHR$(YJ);
2290    GOTO 2160
2300    IF LEN(ZF$)=ZD GOTO 2350
2310    IF ZE$="," THEN ZE$=";"
2320    ZF$=ZF$+ZE$
2330    PRINT @ZC,ZF$;
2340    GOTO 2160
2350 IF ZE=0 THEN ZF=VAL(ZF$)
2360 RETURN
```

Figure 5-2. INPUT Module Listing

The first thing that INPUT does is to set variable XX to 0. Variable XX is the "flag" variable that indicates whether the CLEAR (Model I/III) or "up arrow" (Model II) key has been pressed to terminate the current operation. Variable XX will be returned as a 0 if the key has not been pressed, or as a 1 if the input operation has been terminated.

A check is then made to make certain that the screen location variable ZC is within the limits of 0 through 1023 or 1920. If ZC is not a screen location within these limits, INPUT performs a STOP so that the user can correct his program for a valid value in ZC.

A check is also made for the length of the input string, held in variable ZD. A STOP occurs if the length is not between 1 and 255 characters.

CALL:

```
21140    XB$="FIELD # TO MODIFY":XB=0:GOSUB 11000:IF XX=1 GOTO 21240
21150    IF XC=0 GOTO 21230
21160    IF XC<1 GOTO 21140
21170    GOSUB 9000
21180    'INPUT NEW STRING FOR FIELD
21190    ZC=ZS(XC):ZD=ZR(XC):ZE=1:GOSUB 2000 :IF XX=1 GOTO 21240
21200    ZW$(XC)=ZF$
21210    GOTO 21140
```

INPUT CALL

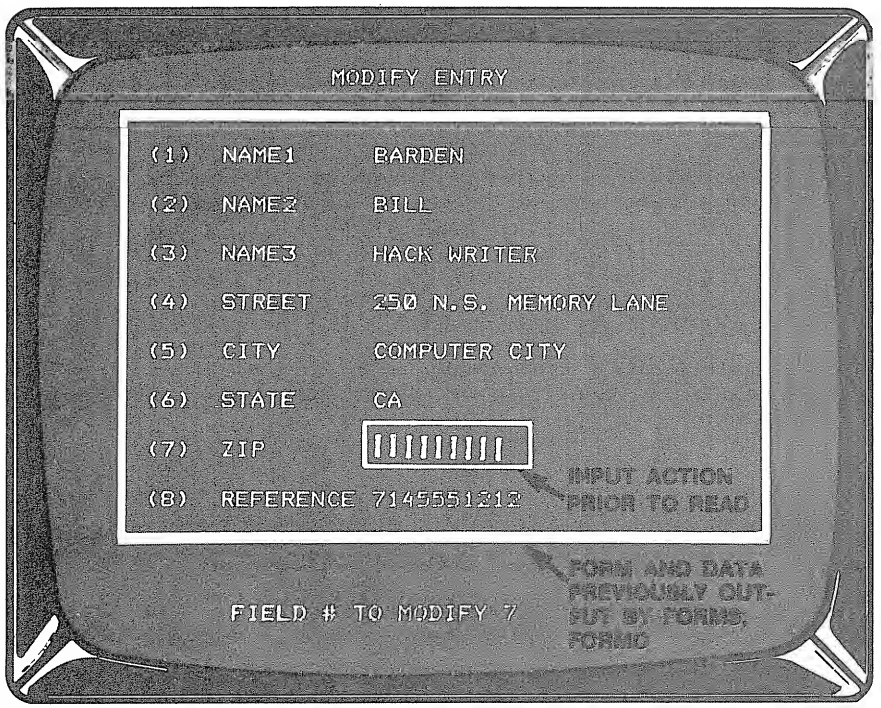


Figure 5-3. INPUT Module Call and Action

Next, the input field is displayed at screen location ZC by performing a PRINT @ ZC. The STRING$ function is used to display a string of ZD "field prompt" characters. The field prompt character is defined by variable YJ and is either a vertical bar (Model I/III) or an asterisk (Model II).

Next, string variable ZF$ is cleared to a "null" (zero length) string. Variable ZF$ will hold the input string on RETURN from INPUT.

The double-indented code (line 2160 through 2200) is the INKEY$ loop. It performs the INKEY$ function by setting variable ZE$ equal to INKEY$. If no key was pressed (ZE$=INKEY$=" ") the code loops back to INKEY$ once again.

The two PRINT statements in the loop are used to flash a "cursor" on and off. String variable ZF$ is the current string that has been entered at any point. The PRINT @ ZC+LEN(ZF$),CHR$(YJ); statement displays the YJ prompt character at the current character position in the entry field. Three statements later, a blank (" ") is displayed at the same character position. It takes some time to execute the four statements. The effect is that during the time no key is being pressed an alternating prompt character and blank are displayed. This appears as a blinking prompt character at the character position for the input.

If a key is pressed, ZE$ is "non-null." The single-indented loop between 2210 and 2340 is then entered. There are five actions taken in this loop, dependent upon the "current" character in string variable ZE$.

Normal Character

If the character is greater than the YK character (ZE$>CHR$(YK)), line 2300 is executed. This will happen if the character is a "normal" alphanumeric or special character. The code at line 2300 first checks to see that the current input string in ZF$ is not equal to the maximum input length. If it is, a GOTO 2350 results, which we'll talk about shortly.

If the length is less than the maximum, the current character in ZE$ is added to (appended to) the ZF$ string by ZF$=ZF$+ZE$. The new string is then displayed at ZC by the PRINT ZC,ZF$;. This amounts to displaying all previous characters plus the new character from ZE$. A jump is then made back to the "inner loop" at 2160.

ENTER Character

If the ZE$ character is an ENTER character (ZE$=CHR$(13)) then the current key being pressed is an ENTER. This marks the end of the input and the end processing at line 2350 is entered. Up to this point each character has been echoed to the screen and ZF$ contains the entire input string that has been entered from the keyboard.

Terminate Operation Character

If the character in ZE$ is a CLEAR (Model I/III) or "up arrow" (Model II), then the input operation is to be terminated. (This might happen, for example, if the input was started and the user discovered that he was entering data for the wrong field. There must be a means to stop the operation and restart.) If this happens, ZE$=CHR$(YK), variable XX is set to 1, and the RETURN is executed. Variable XX is the flag that indicates a "terminate operation" has occurred.

Backspace Character

If ZE$=CHR$(8), then a backspace (Left Arrow or BACKSPACE) key has been pressed. If ZF$=" " at this point, either no character has been entered, or

backspaces have been done to the start of the field. In either case, a GOTO 2160 for the next character is performed.

If ZF$ contains a partial input string, then the last character entered must be deleted and the modified string displayed. The string minus the last character is put into ZF$ by doing a ZF$= LEFT$(ZF$,LEN(ZF$)-1). The LEFT$ command takes all characters of ZF$ except the last. The modified string is then displayed at ZC by the PRINT @ZC, ZF$+CHR$(YJ);. The CHR$(YJ) replaces the deleted character by the field prompt character. A GOTO 2160 then reenters the INKEY$ loop.

None of the Above

If the ZE$ character is none of the above, then the key pressed is not a valid character for the GPM input. It is dropped into the bit bucket (on the floor behind the machine) and the program goes back to look for the next INKEY$ character.

The two statements at the end of INPUT RETURN to the calling program after first checking the ZE variable. If ZE=0 then the input string represented a numeric string, such as "123." For convenience, the numeric string is converted from its ZF$ string form to an actual numeric value by ZF=VAL (ZF$).

In any case, ZF$ contains the input string entered during the INPUT operation. The display shows all valid entered characters, except of course, for deleted characters and the ENTER.

FORMI Module Operation

This module is shown in Figure 5-4. It operates in conjunction with the FORMS and INPUT modules to read in the character strings for the form fields. The FORMS module first outputs a form to the screen. The FORMI is then called to allow the user to "fill in" the fields of the form. FORMI calls the INPUT module to read in and display the fields. A sample call and action for FORMI is shown in Figure 5-5.

```
8500 GOTO8570 'FORMI
8510 '*************************************************************
8520 ' THIS IS THE FORM INPUT MODULE. AFTER A FORM HAS
8530 ' HAS BEEN OUTPUT, THIS MODULE READS IN THE FORM FIELDS.
8540 '      INPUT: NO PARAMETERS
8550 '      OUTPUT: INPUT STRINGS IN ZW$(1) - ZW$(N)
8560 '*************************************************************
8570    FOR ZI=1 TO ZQ
8580    ZC=ZS(ZI):ZD=ZR(ZI):ZE=1
8590    GOSUB 2000 :IF XX=1 GOTO 8620
8600    ZW$(ZI)=ZF$
8610    NEXT ZI
8620 RETURN
```

Figure 5-4. FORMI Module Listing

CALL:

```
22220 ZP=55:ZQ=4:ZP$(0)="PRIMARY DISPLAY/PRINT"
22230 ZP$(1)="START# (ENTER IF NOT KNOWN)"
22240 ZP$(2)="START ENTRY (ENTER IF NOT KNOWN)"
22250 ZP$(3)="END# (ENTER IF NOT KNOWN)"
22260 ZP$(4)="END ENTRY (ENTER IF NOT KNOWN)"
22270 ZR(1)=3:ZR(2)=15:ZR(3)=3:ZR(4)=15:GOSUB8000
22290 GOSUB8500:IFXX=1GOTO22950
```

FORMS CALL

FORMI CALL

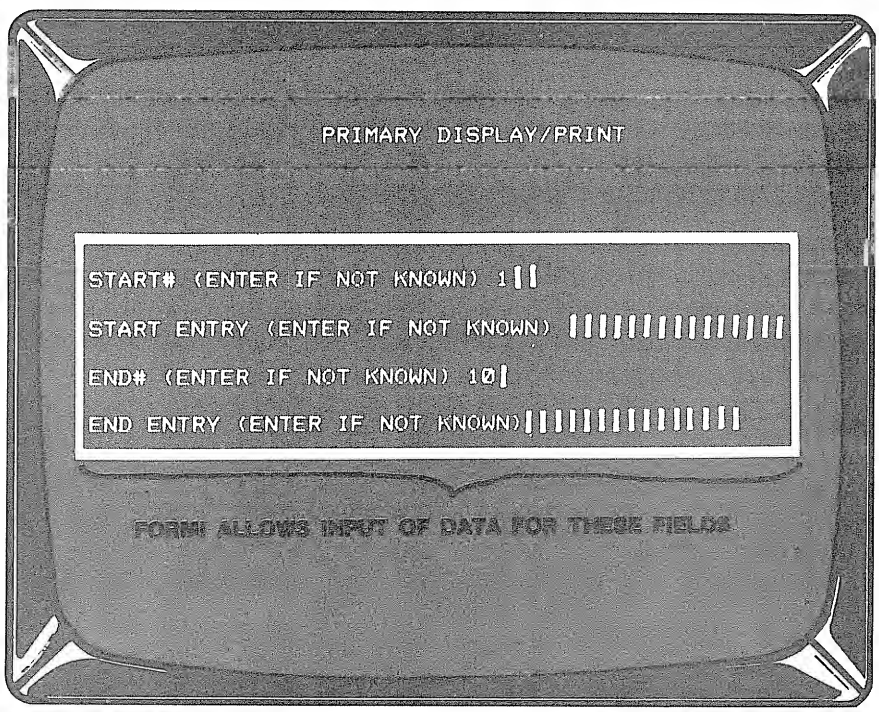


Figure 5-5. FORMI Module Call and Action

After FORMS has output the form, the ZS array contains the screen locations for each of the screen fields. Variable ZQ contains the number of fields on the form. For example, suppose that the form shown in Figure 5-6 has been displayed on the screen by FORMS. ZQ is set to 4 before the FORMS call and still retains that value before the FORMI call. After the FORMS operation ZS has been filled as follows:

ZS(0)=unchanged
ZS(1)=219 Location of field 1
ZS(2)=283 Location of field 2
ZS(3)=347 Location of field 3
ZS(4)=411 Location of field 4

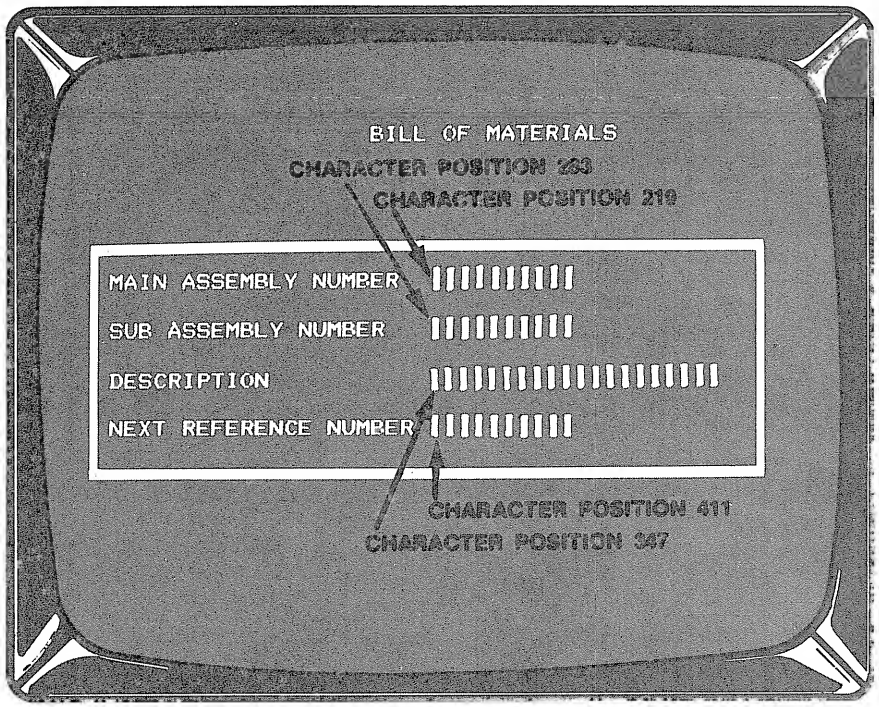


Figure 5-6. Field Definition

FORMS is also called with array ZR containing the length of each entry field. This array is maintained (not changed) so that when FORMI is called, the ZR array contents can be read for the FORMI call.

FORMI uses the locations of each field entry area in the ZS array to call the INPUT module. The number of calls made to INPUT at line 2000 equals the number of fields. Each call is made with ZC set to the location of the next field and with ZD set to the maximum length of the field. Variable ZE is set to 1 as numeric input is not required.

After each call to INPUT, string variable ZF$ contains the input string. This is stored in the string array ZW$ after the GOSUB 2000 call.

Variable ZR is used as a general purpose "working" variable to step from 1 to the number of fields.

PROMPT Module Operation

Operation of the PROMPT module (Figure 5-7) was discussed earlier in the previous chapter. In this discussion we'll talk about the INKEY$ operation. PROMPT in this mode is used to output a system prompting message such as NUMBER OF RECORD? and to read in the corresponding reply. The message is displayed in the prompt message area defined by location YE. A sample call and action is shown in Figure 5-8.

```
11000 GOTO 11110 'PROMPT
11010 '*************************************************************
11020 ' THIS IS THE "PROMPT" MODULE. IT OUTPUTS A GIVEN MESSAGE
11030 ' AT LAST LINE AND READS IN A USER STRING OR NUMERIC
11040 ' RESPONSE.
11050 '      INPUT: XB$=MESSAGE TO BE OUTPUT
11060 '             XB=0 IF NUMERIC RESPONSE,=1 IF STRING,
11070 '               =2 IF YES OR NO RESPONSE,=3 NO RESPONSE
11080 '      OUTPUT:XC$=STRING RESPONSE OR "Y" OR "N"
11090 '             XC=NUMERIC RESPONSE OR 0 IF ENTER
11100 '*************************************************************
11110 XX=0
11120 PRINT @ YE,XB$+"                          ";
11130 IF XB=3 GOTO 11280
11140 XC$=""
11150    XI$=INKEY$:IF XI$="" GOTO 11150
11160    IF XI$>CHR$(YK) GOTO 11200
11170    IF XI$<>CHR$(YK) GOTO 11190
11180    XX=1: GOTO 11290
11190    IF XI$=CHR$(13) GOTO 11230
11200    XC$=XC$+XI$
11210    PRINT @ YE+LEN(XB$)+1,XC$;
11220    GOTO 11150
11230 IF XB=0 THEN XC=VAL(XC$)
11240 IF XB<>2 GOTO 11290
11250 IF XC$<>"YES" AND XC$<>"Y" AND XC$<>"NO" AND XC$<>"N" GOTO 11120
11260 XC$=LEFT$(XC$,1)
11270 GOTO 11290
11280    FOR XI=1 TO 900:NEXT XI
11290 RETURN
```

Figure 5-7. PROMPT Module Listing

The INKEY$ loop is the indented code from line 11150 through line 11220. String variable XC$ holds the input character string. Initially it is set to a "null" string of " ", or 0 characters.

Variable XI$ is used to hold the input character from INKEY$. If INKEY$="" (no key pressed), a loop is made at line 11150.

There are three cases for entering characters in this INKEY$ code.

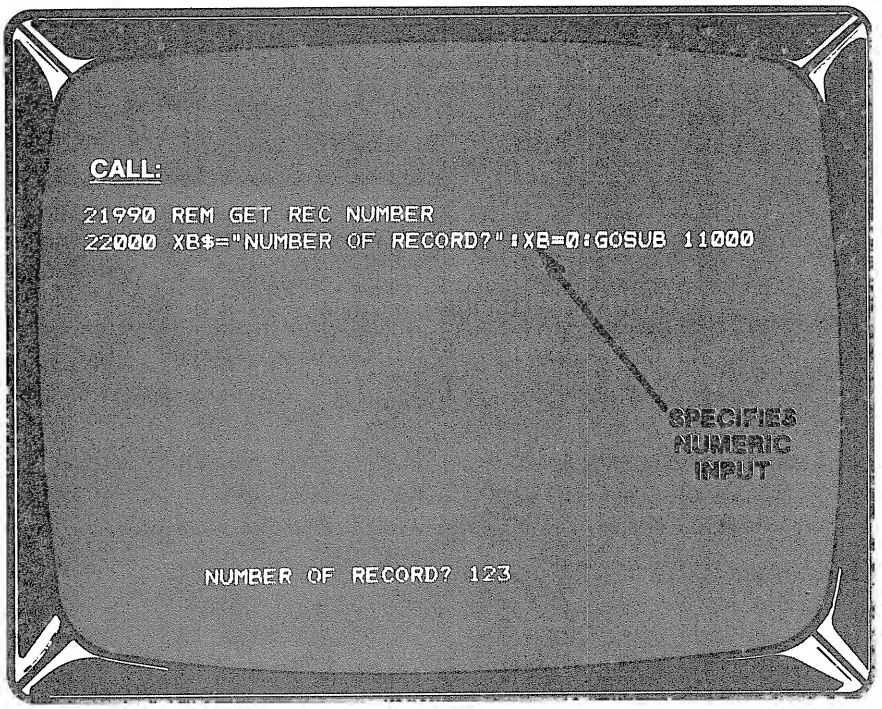


Figure 5-8. PROMPT Module Call and Action

Normal Character

If XI$ is greater than CHR$(YK) then it is an alphanumeric character. In this case the code at line 11200 appends the XI$ character string to the XC$ character string by `XC$=XC$+XI$`. The new string is then printed in the prompt message area, after the prompt message by `PRINT@YE+LEN(XB$)+1,XC$;`. The `YE+LEN(XB$)+1` starts the display at the proper place. The code then goes back to get the next INKEY$ character.

Terminate Operation Character

If the character is a CLEAR (Model I/III) or "up arrow" (Model II) character, then the terminate operation flag variable XX is set to a 1 and a `RETURN` is made. XX was set to a 0 on entry to `PROMPT`. The terminate operation character is used to provide a means to "restart" if the user finds himself in the middle of a prompt sequence when he does not want to be there.

ENTER Character

If the character is an ENTER character (XI$=CHR$(13)), then the input operation is over. A transfer is then made to the end processing portion of the `PROMPT` module.

End Processing

When line 11230 is executed, XC$ holds the string response to the prompt message, except for the case when no input is required (variable XB=3 on entry).

If input variable XB=0, the input response is numeric. The character string in XB$ is converted to a numeric value by `XC=VAL(XC$)` and the next statement causes a `RETURN`. The `RETURN` is made in this case with XC$ containing the string response and XC equal to the numeric value of the string.

If input variable XB=1 (string response specified) a return is made with XC unchanged.

If input variable XB=2, a `YES` or `NO` response is called for. A `YES` or `NO` is always changed to a `Y` or `N` for easy comparison in the main routine. In this case line 11250 looks for a `YES`, `Y`, `NO`, or `N`. If none of these are equal to XC$, the program loops back to repeat the entry. If any of the four is present, XC$ is set equal to the leftmost character of XC$ by `XC$=LEFT$(XC$,1)`. A `RETURN` is then made with `XC$=Y` or `N`.

Chapter Six
Data Storage Using the GPM

We're going to cover an important aspect of the General Purpose Modules in this chapter, **data storage**. First we'll discuss the problem of data storage in general terms, then we'll talk about the philosophy used in the GPM, and finally we'll discuss how each of the GPM modules operates.

The techniques of data storage are sometimes called **data structures**. Data structures include such things as DATA statements, lists, and arrays. Also included in this area are the **types** of data to be stored — strings, integer values, single-precision values, and so forth.

The Problems of Data Storage

At first glance, it doesn't seem like there should be a problem in storing data. We know from our BASIC reference manuals about variables in general, DATA statements, **one-dimensional arrays, or lists**, and more complex arrays, including string arrays. However, the choice of what type of storage we use directly affects the program operation as far as speed, memory storage, disk or cassette operations, **sorting** the data, **searching** for values, and so forth. Let's take a look at some of the storage alternatives that BASIC offers and discuss their strengths and weaknesses.

Variables

The first type of storage that a user comes in contact with is **simple variables**. This is not an aspersion on the variable's intelligence, just a qualifier that says the variable is not part of a complex variable such as an array.

A simple variable, as you know, is defined by any two letter name, the first of which must be alphabetic. You may also know that there are a number of variable types:

- Integer variables
- Single-precision variables
- Double-precision variables
- Double-precision variables with scientific notation
- String variables

Integer variables can only hold integer values from -32768 to +32767. Single-precision, double-precision, and double-precision with scientific notation variables can hold fractional values and large numbers at some sacrifice of accuracy — for example, 32.5567899 is printed as 32.5568, as a single-precision variable. String variables are used to hold character strings.

At this point, you've probably worked with some single-precision and string variables, and we won't dwell on their descriptions or on descriptions of the double-precision variables, as they are similar in concept.

Variables are usually used to store intermediate results in the course of a program. In a large program you'll have many intermediate results, and a correspondingly large number of variables.

The advantages of using simple variables are these:

- They're easy to use.
- There are a lot of them — 936, not counting names that are identical except for data type suffixes, such as AA and AA%.
- In some cases the name can be used as a **mnemonic** device, such as variable "CT" for Current Total.

You could construct an entire program with just simple variables. However, it would suffer from some drawbacks. Some of the disadvantages of simple variables are:

- Each separately named variable must be located by the BASIC interpreter. With a lot of variables, this takes a great deal of time. (A second is an eternity both in microcomputer timing and when the dentist says "I'll be done in a second!")
- Similar types of data are not grouped together. If variable AA$ holds the account number, variable AB$ holds the company name, variable AC$ holds the address, and so forth, it may become quite tedious to work with collections of these variables to represent one set of data.
- More memory storage than necessary may be used to hold data. Each numeric variable takes up from five to eleven bytes, and with a large number of variables, this may become a significant factor in total program size.

DATA Statements

Another type of storage is by means of DATA statements. You've probably performed a simple program like

```
100 A=0
110 FOR X=1 TO 10
120 READ C
130 A=A+C
140 NEXT X
150 PRINT ''TOTAL IS:'';A
160 DATA 2,15,45,67,34,56,5,1,5,100
```

This program READs a value from the DATA list into C, and then adds the value to a subtotal in A, finally printing the total.

The DATA list in statement 160 is an easy way to list numeric values for processing. It is fine for simple programs and for student exercises, but has many weaknesses:

- The DATA must be in the program text. External data cannot be entered into the DATA list.

- The data must be read **sequentially** from beginning to end. It is possible to start at the beginning of the list by using the RESTORE command, but it is not possible to retrieve the third, or fifth, or twenty-third item, without going through a READ for each item.
- The RESTORE positions a "pointer" to the beginning of the list, but there is no way to position the pointer to a random spot in the list.
- Separate DATA statements form one collective list, rather than separate groupings of data related to different functions.

Because of the above limitations, DATA statements are seldom used in larger applications programs, except possibly for "initialization" of variables at the beginning of a program.

Arrays

Arrays are a number of similar **types** of variables (such as integer or double precision or string variables) grouped together under one name. Items within the array are referenced by the array name and **element** number. For example, BOOK would be displayed here from the EX$ array containing BELL, BOOK, and CANDLE:

```
100 DIM EX$(3)
110 EX$(1)="BELL"
120 EX$(2)="BOOK"
130 EX$(3)="CANDLE"
140 PRINT EX$(2)
```

As you know from your BASIC reference manuals, arrays may be **one-dimensional**, such as the one above, or **multi-dimensional**. A two-dimensional array might represent a chessboard by

```
100 DIM CB(7,7)
```

There are a number of advantages to arrays:

- Similar types of data can be conveniently grouped together.
- Total memory storage is less than storing a separate variable for each item in the array.
- Total overall processing time may be reduced in storing or retrieving data over working with separate variables.

Of course, there are a number of disadvantages, too:

- Each item in the array must be referenced by its number or "coordinates," and not by a name, and this may become confusing.
- Since the array is (usually) one huge block of data, it may be difficult to **sort** the data in the array to put it in some sort of logical order, either alphabetic or otherwise. The same problem occurs in searching for data.

- Multi-dimensional arrays are hard to use, as it is tedious to compute the location of the data within the array.

Execution Speed and Memory Storage

It seems suprising to many microcomputer users that many applications programs, business and otherwise, are too slow and too large. (An old programming axiom in use before the days of "virtual storage" states that programs always occupy 200 bytes more memory than is available . . .) The very fact that BASIC is a high-level language and easy to use, however, causes a degradation in both speed and storage efficiency.

With a Disk BASIC Model I, we have about 48K (49152) bytes of RAM available initially for user program and data storage on a system with maximum memory. About 10K of the 48K is dedicated to TRSDOS and BASIC software, leaving 38K. That 38K must be divided up between the user program and data storage. If we assume that we are working with 200 **records** of data with about 60 bytes apiece, that means that 12K of memory must be available to hold all the records (of course, we could hold portions of the records at the expense of a lot of disk activity). That leaves 24K to hold the user program, miscellaneous variables, arrays, and string working area. As a typical program might contain 400 lines at 40 characters per line, we now have 8K for "working storage." You can see that there is not an unlimited amount of memory available by any means.

How about speed? Computers are supposed to be able to calculate in millions of operations per second. It's true that microcomputers can add hundreds of thousands of numbers per second; the TRS-80 Model I can add at the rate of about 445,000 operations per second, for example.

However, the TRS-80s have to execute many instructions just to **interpret** a single line of BASIC code. The rate at which **BASIC lines** may be executed ranges from less than 50 to 800 lines per minute. If the BASIC program is performing operations such as **sorting** data, delays in processing may not only be noticeable to the user, they may be interminable.

Data Storage in the GPM

The basic method of data storage we've used in the General Purpose Modules is that of a **one-dimensional string array**. Data is held in the form of a number of string **entries** in the array as shown in Figure 6-1, which holds entries for a small file.

The string array has the name of XA$, chosen in the "X, Y, Z" range of letters that we're using for the GPM. The size of the array is based upon how much memory is available after loading the GPM, the user's application program that uses the GPM, and CLEARing a portion of memory for strings.

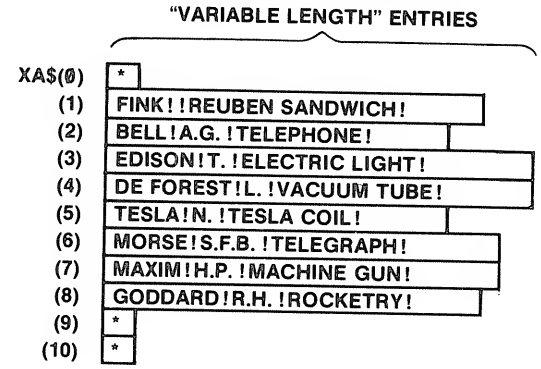


Figure 6-1. String Array Example

Each entry in the XA$ array may be from 1 to 255 characters in length, so that the entire array holds a number of **variable length** entries.

Each entry in the XA$ array is further subdivided into **fields**. A field is any logical subgroup of an entry, dependent upon the application program. In the example of Figure 6-2, for example, the fields are last name, initials, and invention. Each field is terminated (or in computer jargon, **delimited**) by an exclamation point (!). When a field contains no characters, two exclamation points will be together, as shown in the figure.

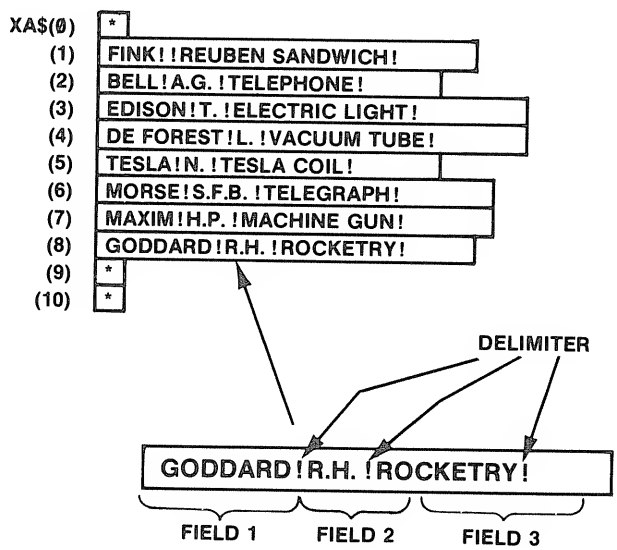


Figure 6-2. Fields in Entries

You'll notice that the entries in the array appear to follow no alphabetical order. Actually, there is an order to the array, and it's held by a second array called XA%. XA% is an **integer** array made up of entries that are each two bytes long as shown in Figure 6-3. There is one entry in XA% for each entry in XA$. We'll talk about the relationship of the XA% and XA$ arrays shortly.

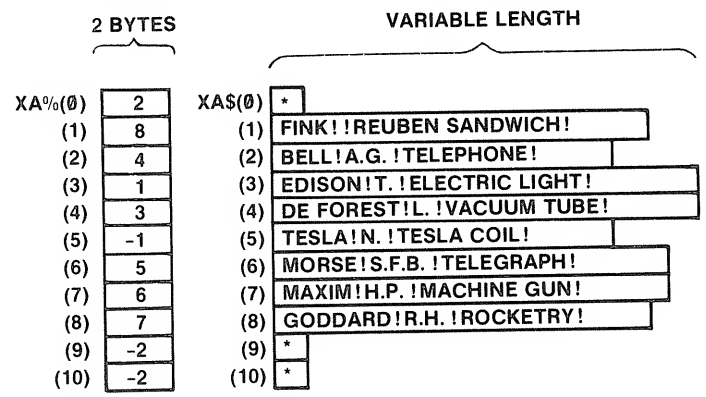


Figure 6-3. XA% and XA$ Arrays

The second method of storage we've used in the GPM is by simple variables. We've made all of the variables in the General Purpose Modules **integer variables** to cut down on storage requirements. We've used as **few** variables as possible to avoid having a long list of variables that the BASIC interpreter has to search. All variables in the GPM use the naming conventions of X, Y, and Z, and these are variable names that your application program should not use.

Sorting and Searching

Before we discuss more on how the array is ordered, let's first talk in general terms about **sorting** and **searching**. Sorting refers to putting long lists of data in a logical order, such as in alphabetizing. Searching refers to finding an entry within a sorted (or unsorted) list.

Let's assume that all data will be held in the one-dimensional XA$ string array. Suppose that we have 200 entries in the array and want to find a particular one. We could input a search "key" and proceed as follows:

```
100 INPUT A$
110 FOR I=1 TO 199
120 IF A$=XA$(I) GOTO 150
130 NEXT I
140 PRINT ''NOT FOUND'':STOP
150 PRINT ''FOUND AT ''; I:STOP
```

The above code searches the array from bottom to top (or, for you southern hemisphere readers, from top to bottom) for the specified entry of A$. If the

entries in XA$ were each 30 characters long, the **average search** would find the entry at entry 100 and take about one second; the **worst case** search would be about twice the average search time. Although this seems very fast, when other operations must be done during the search and when the size of the array grows, the search becomes slower and slower.

Because searches are very time consuming for **unordered** data, almost all long arrays are **ordered** — either in alphabetical, numeric, or some other order. When arrays are ordered, faster search algorithms (such as the **binary search**) can be used.

The order most commonly used for string arrays is **alphanumeric order**. In this order, the entries are ordered in "phone book" fashion in this priority: space, !, ", #, $, %, &, ', (,), *, +, comma, -, period, /, 0-9, colon, semicolon, <, =, >, ?, @, A-Z, a-z. Typical entries with this ordering sequence would be

```
!EXPLETIVE DELETED!
$AVE YOUR PENNIES
$ave your pennies
SMITH, JOHN
SMITH, TOM
ZZYYXX
```

How does an array get ordered? One method is to simply search for an **insertion point** for a new entry and move everything down, as shown in Figure 6-4. If we wanted to insert a new entry at the 100 element of the XA$ string array, for example, we could do:

```
100 FOR I=100 TO 198
110 XA$(I+1)=XA$(I)
120 NEXT I
130 XA$(100)="NEW ENTRY"
```

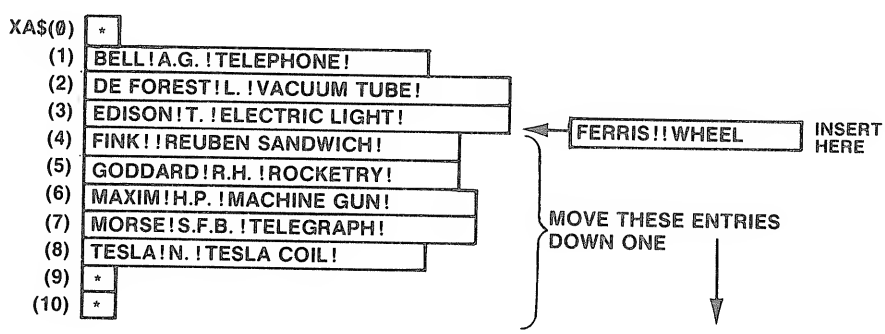


Figure 6-4. Inserting a New Entry — One Alternative

As a matter of fact, though, this method is **much** too time consuming. Although this simple case takes about one second, the speed increases greatly for more complicated code and larger arrays. Because this simple approach is too slow, sophisticated techniques of sorting such as the **Shell-Metzner** sort have been developed.

It's somewhat of a paradox — data can't be found without ordering arrays, and yet we must go to elaborate sorting techniques to order the data in a reasonable time!

Ordering in the GPM

We've tried to develop an efficient means of sorting and searching in the General Purpose Modules. Like every method, it's a compromise. The scheme used is called a "linked list." It's nothing new, but it does seem to work well for string arrays.

Here are its advantages:

- It permits easy "insertions" into the array
- It permits easy deletions of entries
- It will work easily with variable-length entries with any number of fields

Here are its disadvantages:

- Searches for entries are done from the beginning of the array
- It is moderately complex

We've already described the XA$ array and how it holds variable-length string entries with field subdivisions. Now let's see how the XA% array relates to the XA$ in the linked list structure.

The XA% array is an **integer** array. It holds the same number of entries as are in the XA$ array. XA%(1) corresponds to XA$(1), XA%(2) to XA$(2), and so forth as shown in Figure 6-5. The order of the XA$ array is maintained by the **pointers** in the XA% array.

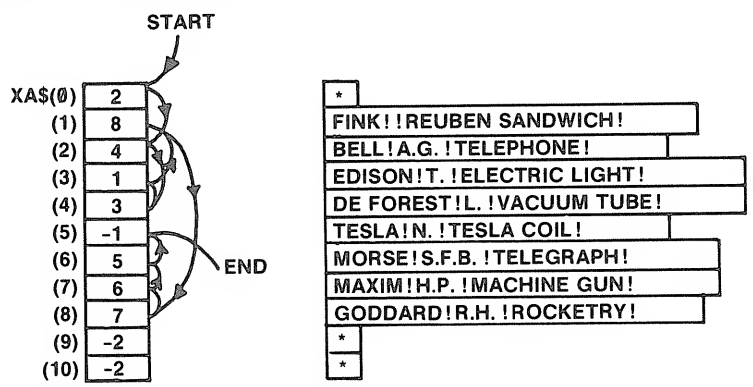


Figure 6-5. Using the XA% Array to Order XA$

The first element of XA%, XA%(0), holds a pointer value that represents the next item number in the ordered list. In Figure 6-6, XA%(0) holds 2, which means that XA$(2) contains the first entry of the XA$ array. XA% entry 2 holds a pointer to the next entry. In this case it is 4, which means that the second entry is located at XA$(4). This chain continues, "link" after "link" until the last entry in the array is found. The last entry has a value of -1, indicating that there are no more entries in the list. As you can see from Figure 6-5, starting with XA%(0), it's easy to trace a chain of items by going forward in the list.

With this structure, we can eliminate the time-consuming "overhead" of moving around large amounts of string data. The entries in XA$ stay where they are; only the pointers in XA% are adjusted.

Deleting an Entry

To delete an entry, all that has to be done is to remove the link from the chain and adjust the pointers in XA%. This process is shown in Figure 6-6 for a deletion of one entry in XA$. The deleted entry in XA$ is "blanked out" by replacing the XA$ entry with the string "*". This marks the XA$ entry as "unused" so that a new entry can use the XA$ position. At the same time the corresponding XA% entry is marked unused by setting it to -2.

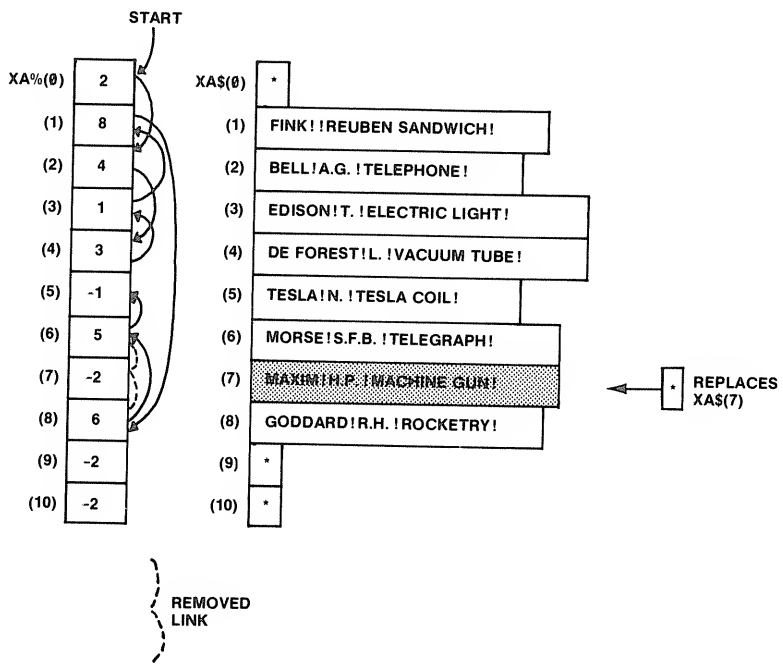


Figure 6-6. Deleting an Entry Using a Linked List

Adding an Entry

The process for adding an entry is shown in Figure 6-7. To add an entry, the "chain" is broken and the previous pointer in XA% is set to the number of the added entry position. The pointer in XA% corresponding to the added entry position is loaded with the "next" entry.

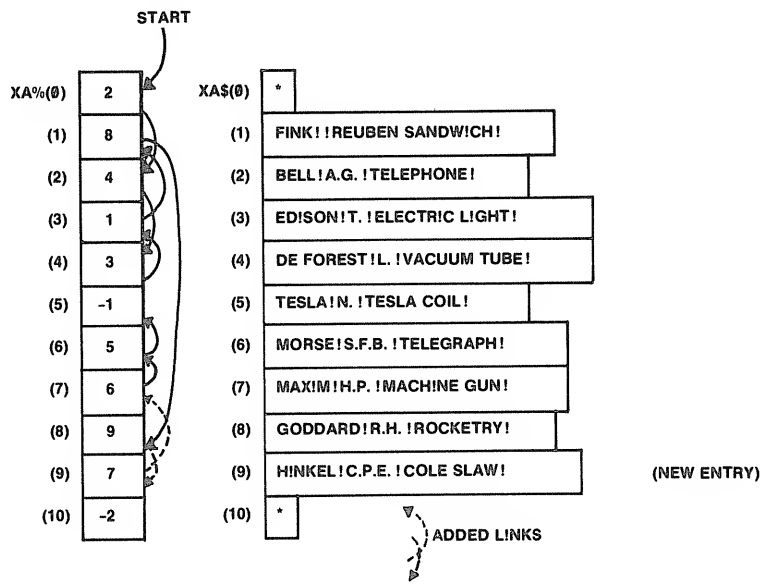


Figure 6-7. Adding An Entry Using a Linked List

Modifying an Entry

Modifying an entry might change the entry such that it is out of order. Changing AABLE to ABLE, for example, would mean that ABLE must go between AARDVARKE and ACTIVE. For this reason, a modify may be handled by a deletion and then an add, as shown in Figure 6-8.

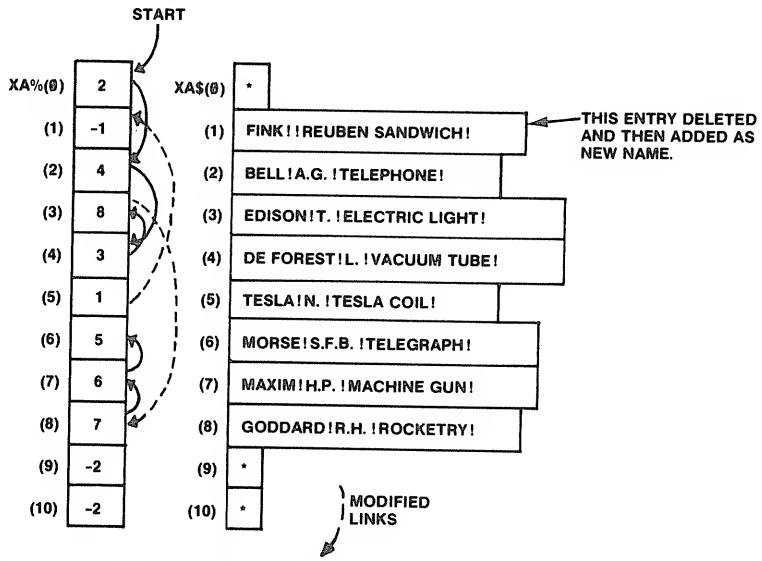


Figure 6-8. Modifying an Entry Using a Linked List

Initial Conditions

Initially all entries of XA$ are filled with "*" and the XA% entries are filled with -2 to mark them "unused." XA%(0) is a special case. Initially it is set to -1, as there are no entries in the XA$ array. This is compatible with the logic that says "if the pointer value to the next entry is a -1, you have come to the last entry in the list." Initialization values are shown in Figure 6-9.

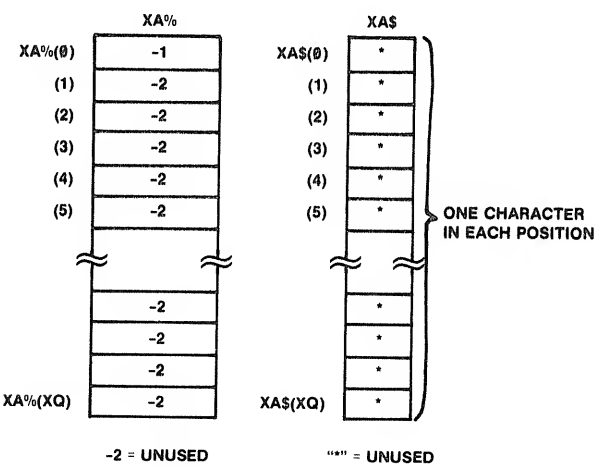


Figure 6-9. Initialization of GPM Arrays

AINIT Module Operation

The AINIT module is shown in Figure 6-10. It is used to initialize the XA$ and XA% arrays and to set the "system variables" in the Y letter range based on the type of system used.

```
10500 GOTO10610 'AINIT
10510 '*****************************************************
10520 ' THIS IS THE INITIALIZE ARRAYS MODULE. IT MUST BE CALLED
10530 ' BEFORE ANY ARRAY PROCESSING IS DONE. IT SETS UP THE
10540 ' XA% AND XA$ ARRAY TO "UNUSED".
10550 '      INPUT:  NO PARAMETERS
10560 '              ***CALLED BY A GOTO***
10570 '      OUTPUT: XA% ARRAY SET TO ALL -2
10580 '              XA$ ARRAY SET TO ALL "*"
10590 '*****************************************************
10600 'CLEAR STRING AREA, DEFINE INTEGER, AND FIND ARRAY SIZE
10610 CLEAR INT(MEM*.85)' ***.85 MAY BE ADJUSTED***
10620 PRINT CHR$(2);
10630 DEFINT X,Y,Z
10640 XJ=0:XK=0:XL=0:XS=0:XU=0:XQ=0:YS=0:XT=0:XI=0:XH=0
10650 ZI=0:ZH=0:XW$="":XD$="":XZ$="":XY$=""
10660 XQ=INT(MEM/52)*4
10670 YE=320: YL=55
10680 DIM XA%(XQ):DIM XA$(XQ):DIM XP(20):DIM ZW$(20):DIMXB%(XQ)
10690 DIM ZA$(11) :DIM ZP$(13) :DIM ZZ(11) :DIM ZZ$(11)
10700    FOR XI=1 TO XQ-3 STEP 4
10710    XA%(XI)=-2:XA%(XI+1)=-2:XA%(XI+2)=-2:XA%(XI+3)=-2
10720    XA$(XI)="*":XA$(XI+1)="*":XA$(XI+2)="*":XA$(XI+3)="*"
10730    PRINT @ YL, XI;
10740    NEXT XI
10750 PRINT @ YL,"      ";
10760 XA%(0)=-1:XA$(0)="*"
10770 PRINT @ 320,"MOD I(1), II(2), OR III(3)?"
10780 XI$=INKEY$: IF XI$="" GOTO 10780
10790 XC=VAL(XI$)
10800 IF XC<1 OR XC>3 GOTO 10770
10810 IF XC=2 THEN YA=80 ELSE YA=64
10820 IF XC=2 THEN YB=24 ELSE YB=16
10830 YC=YA*2:YD=YA*3:YE=YA*(YB-1)+10
10840 IF XC<>2 GOTO 10870
10850 YF=150: YG=150: YH=148: YI=148: YJ=170: YK=30: YL=71
10860 GOTO 10880
10870 YF=176: YG=131: YH=149: YI=170: YJ=138: YK=31: YL=55
10880 ON ERROR GOTO 11500
10890 GOTO 20100' ***CHANGE THIS FOR YOUR SYSTEM***
```

Figure 6-10. AINIT Module Listing

AINIT must be called by a GOTO 10500 before any processing in the user application programs. Note that the call is a **GOTO** rather than a **GOSUB**. The reason for this is that CLEAR actions in the AINIT will destroy any return address from a GOSUB. AINIT returns to location 20100 after it has finished initialization; the application program must expect a return to this address after the AINIT.

The first thing AINIT does is to perform a CLEAR INT(MEM*.85). This statement finds the memory size (the available RAM memory after loading the programs) takes 85% of this memory space, converts the result to an integer, and CLEARS this amount for strings. As you know, the CLEAR clears all variables and sets aside string storage space. The .85 or 85% allocation is purely a "good guess" of the amount of the ratio of string storage space to available memory. The user might try adjusting this value in different applications. If it is too low an "OUT OF STRING SPACE" type error will result.

Next, a PRINT CHR$(2) is done. This code turns off the blinking cursor in the Model II; GPM uses its own blinking cursor. It has no effect in the Models I and III.

Next, variable names in the X, Y, and Z range are declared as integers (DEFINT X,Y,Z). All GPM variables will hold values from -32768 to +32767 so the integer range is adequate. Integer variables take only two bytes of storage each and their processing is much faster than other numeric variables.

Next, a number of critical variables are declared. The only purpose of this statement is to store the variables at the beginning of the variable storage area, making the retrieval of these often used variables as fast as possible. This speeds up the overall execution of the program.

Next, variable XQ is computed. Variable XQ is the dimension of the XA$, XA%, and XB% arrays. It is a multiple of four (plus one byte) since later processing initializes the arrays by STEPping in increments of 4. The size of the arrays is based on the current memory divided by 13, an arbritrary figure. If your array entries will be a small number of characters, use a smaller divisor; if your array entries will be a larger number of characters, use a larger divisor. The 13 divisor is based on entries of about 60 characters.

Next YE (prompt message location) and YL (activity field start) are temporarily initialized for AINIT actions.

The next statements allocate major GPM arrays.

The code that is indented (lines 10700 through 10740) is a FOR...NEXT...STEP loop that sets all XA$ entries to "*" and all XA% entries to -2 (unused). After each four entries have been initialized, the current number is displayed at YL. This denotes "activity" and reassures the user that the program is still running. The next two statements clear the activity area and set XA$(0) to "no entries" and XA%(0) to "*".

The remaining code initializes system variables based on user input of the type of machine being used. Variable XC is set to 1,2, or 3 for the Model I, II, or III. The system variables represent the "parameters" of the system as shown in Table 6-1.

Table 6-1.

VARIABLE	MODEL I/III	MODEL II	DESCRIPTION
YA	64	80	Width of screen
YB	16	24	# lines on screen
YC	YA*2 = 128	YA*2 = 160	Location of 2nd line
YD	YA*3 = 192	YA*3 = 240	Location of 3rd line
YE	970	1850	Location of prompt message
YF	176	150	"Bottom" box character
YG	131	150	"Top" box character
YH	149	148	"Left" box character
YI	170	148	"Right" box character
YJ	138	170	Field prompt character
YK	31	30	CLEAR character
YL	55	71	Location of activity field

Table 6-1. System Variables

The next to last statement sets up the location of the error handling routine. This module (ERROR) is used primarily for "non-catastrophic" errors such as "disk file name not found." Any BASIC error will cause the ERROR module to be entered. The ERROR module will display a user-specified message if desired, or will simply return to BASIC for its error message.

The last statement returns to the user application program at line 20100.

ASRCH Module Operation

The ASRCH module is shown in Figure 6-11. It is one of the most important modules in the GPM, as it searches the XA$ array for either a given entry or a location in the array that will mark where an entry will be inserted (added). A sample call and action are shown in Figure 6-12.

The module is called with string variable XD$ containing the string for the search. XA$ and XA%, of course, contain the entries currently being used.

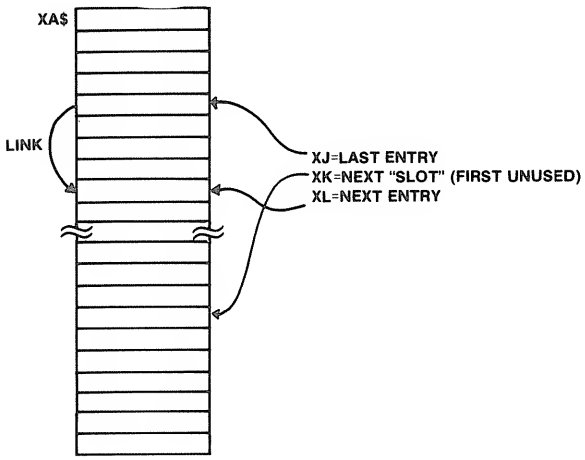
On RETURN ASRCH will pass back these variables:

- XK has the entry number of the next available entry in XA$
- XJ has the entry number of the last entry
- XL has the address of the next entry
- XM is a "found/not found" flag
- XS contains the number of the entry

There are several modes of operation of ASRCH. We'll explain the meaning of each of the modes.

```
5000 GOTO5140 'ASRCH
5010 '*****************************************************
5020 ' THIS IS THE SEARCH ARRAY MODULE. IT IS USED TO SEARCH
5030 ' FOR A GIVEN STRING, EITHER TO FIND THE EXPECTED STRING
5040 ' OR TO FIND THE SPOT WHERE THE STRING SHOULD BE INSERTED
5050 '     INPUT: XD$=STRING FOR SEARCH
5060 '            XA% AND XA$ ARRAYS CONTAIN APPROPRIATE DATA
5070 '     OUTPUT:XJ=INDEX TO ENTRY BEFORE STRING
5080 '            XK=INDEX TO NEXT AVAILABLE SLOT
5090 '            XL=INDEX TO NEXT ENTRY AFTER STRING OR STRING
5100 '               OR -1 IF NEXT ENTRY OUT OF ARRAY
5110 '            XM=0 IF NOT FOUND, 1 IF FOUND, 2=OUT OF MEMORY
5120 '            XS=# OF ENTRY
5130 '*****************************************************
5140 IF XA%(0)<>-1 GOTO 5180
5150 XJ=0:XK=1:XL=-1:XM=0:XS=0
5160 GOTO 5370
5170 'FIRST FIND NEXT AVAILABLE SLOT
5180    FOR XK=1 TO XQ
5190    IF XA%(XK)=-2 GOTO 5240
5200    NEXT XK
5210 XM=2
5220 GOTO 5370
5230 'NOW XK=NEXT AVAILABLE SLOT
5240 XJ=0: XS=1
5250 XL=XA%(0)
5260    PRINT @ YL, XS;"   ";
5270    IF XD$<>XA$(XL) GOTO 5310
5280    'MATCH HERE
5290    XM=1
5300    GOTO 5370
5310    IF XD$<XA$(XL) GOTO 5360
5320    XJ=XL
5330    XL=XA%(XL)
5340    XS=XS+1
5350    IF XL<>-1 GOTO 5260
5360 XM=0
5370 RETURN
```

Figure 6-11. ASRCH Module Listing




```
20580 'NOW CONSTRUCT ONE STRING FROM FIELDS
20590 GOSUB 6500
20600 'NOW SEARCH FOR KEY IN EXISTING FILE
20610 XD$=XY$:GOSUB 5000          <-- ASRCH CALL
20620 'IF NOT OUT OF MEMORY, CONTINUE
20630 IF XM<>2 GOTO 20670
20640 XB=3:XB$="OUT OF MEMORY":GOSUB 11000
20650 GOTO 20680
20660 'NOW ADD ENTRY
20670 GOSUB 7000                  <-- UTILIZES ASRCH PARAMETERS
20680 RETURN
```

Figure 6-12. ASRCH Call and Action

Call Made Expecting to Find XD$

If a call is made with XD$ containing the entire entry string that should be in XA$, then variable XL will contain the entry number of the string and variable XJ will contain the previous entry number. Variable XK will contain the entry number of the first "free slot," which is meaningless in this case. Flag variable XM will be set to a 1 if the entry is found. Variable XS will hold the number of the entry.

If the XD$ string is not found, variable XM will be set to 0, and the remaining variables will be set as in the next case. Figure 6-13 illustrates this mode.

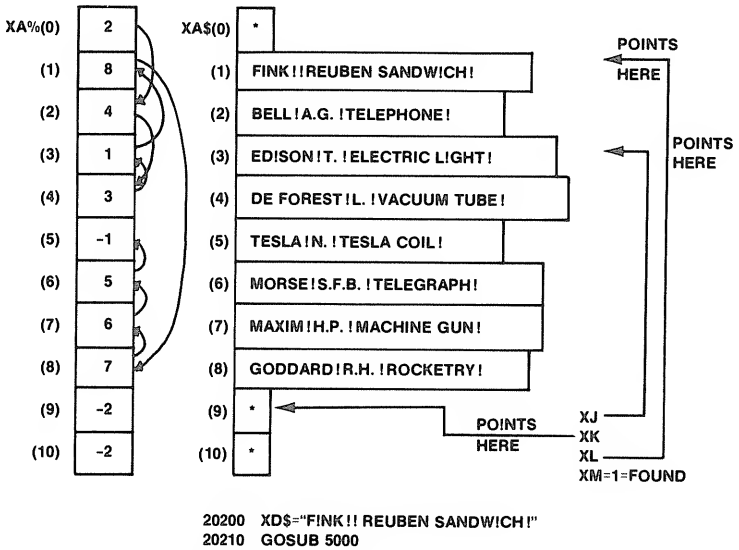


Figure 6-13. ASRCH Call with "Found"

Call Made Not Expecting to Find XD$

This is the mode used for finding the insertion point in the XA$ array for adding a new entry. In this case, flag variable XM will be set to 0 on RETURN.

Variable XK will hold the entry number of the next free entry in XA$. The XD$ string **will not be added to XA$ at this point.**

Variable XL will hold the entry number of the next entry **after** the insertion point, and variable XJ will hold the entry number of the entry just prior to the insertion point. Variable XS will hold the number of the variable associated with the XL entry; this is the count of the entry from the beginning of the list. The "typical" case for a call in this mode is shown in Figure 6-14.

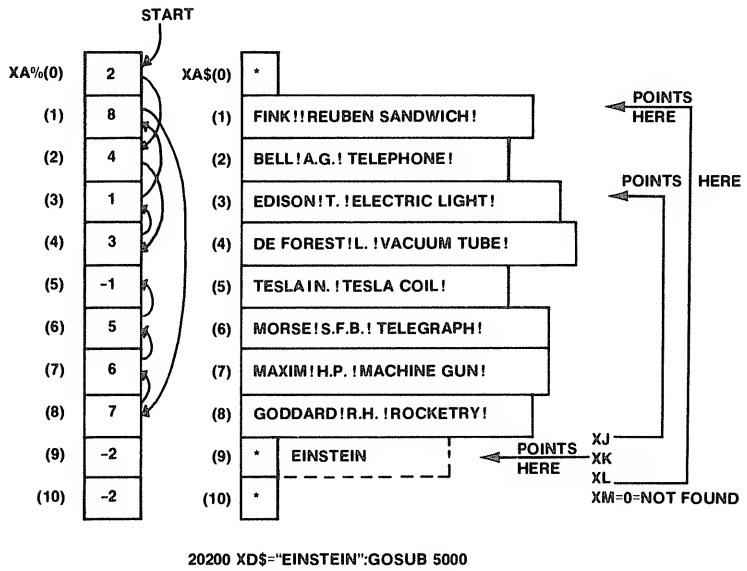


Figure 6-14. ASRCH Call with "Not Found"

There are two "untypical" cases here. If the XA$ array is empty, XJ contains a zero for the XA%(0) entry and XL is set to -1 to indicate that there is no "next" entry. All other variables are set as before. If the XD$ string is "greater" than the last entry, the insertion point is at the end of the list. In this case XL contains a -1 to indicate that there is no "next" entry.

General Operation

ASRCH first checks for an empty list by testing XA%(0) for 0. If the list is empty, XJ is set to 0, XK to 1, XL to -1, XM to 0, and XS to 0. A RETURN is then made.

The first set of indented code searches for the first unused entry. Every unused entry is set to -2 in XA% either from initialization, or as entries are deleted. Variable XK is set to the entry number of XA% that contains the first -2. If the loop is completed, no unused entry has been found in the entire list, and an "out of memory" return is made (XM=2).

The code at line 5240 is entered if the array is not empty. XJ is set to 0 and XS to 1. Variable XL is then loaded with the "next" value from XA%(0). From this point, the loop at 5260 through 5350 is executed. It continues until the XD$ string is found, until an insertion point is found, or until the entire array is scanned (insertion point at end).

The first action in the loop is to print the current number at the activity area. The search may be seconds in some cases, and this indicates activity to the user.

Next, a comparison of XD$ and XA$(XL) is made. If they are equal, the entry has been found (there is a "match"). In this case XM is set to 1 and a RETURN is made.

If they are not equal, a test is made for XD$ being less than XA$(XL). If this is the case, an "insertion point" has been found, flag XM is set to 0 for "not found" and a RETURN is made.

If they are equal or XD$>XA$(XL), then the next entry in XA$ must be examined. XJ (previous) is set to XL (next). XL is then set to the next entry number from XA%(XL). The count of entries in XS is incremented by one. At this point XL contains the entry number of the next entry. If this entry number is -1, the end of the list has been reached, flag XM is set to 0, and a RETURN is made. If the next is not -1, the loop continues.

AADD Module Operation

The AADD module is shown in Figure 6-15 in all its complexity. It adds an entry to XA$. It takes the XL, XJ, and XK variables values from an ASRCH call immediately preceding. XD$ is copied into the next available entry in XA$ by XA$(XK)=XD$. The next available value in XA% is then set to the next entry number by XA%(XK)=XL. Finally, the previous value in XA% is adjusted to hold the added entry number by XA%(XJ)=XK.

```
7000 GOTO 7070 'AADD
7010 '*************************************************************
7020 ' THIS IS THE ADD ENTRY MODULE. IT ADDS AN ENTRY TO THE
7030 ' XA$ ARRAY AND SETS APPROPRIATE XA% POINTERS
7040 '      INPUT: XJ,XK,XL SETUP FROM SEARCH MODULE
7050 '      OUTPUT:ENTRY ADDED
7060 '*************************************************************
7070 XA$(XK)=XD$
7080 XA%(XK)=XL
7090 XA%(XJ)=XK
7100 RETURN
```

Figure 6-15. AADD Module Listing

ADEL Module Operation

The ADEL module is shown in Figure 6-16. It deletes an entry from XA$ based on ASRCH call parameters just prior to ADEL. Since XK (first unused entry number) is not required, it is used as a temporary variable. XK is first set to the

next entry number from the entry to be deleted. The previous value in XA% is then set equal to this entry number by XA%(XJ)=XK, "breaking the link." The last two actions set the XA% entry and XA$ entry to "unused."

```
9500 GOTO9570 'ADEL
9510 '*****************************************************
9520 ' THIS IS THE DELETE ENTRY MODULE. IT DELETES AN ENTRY FROM
9530 ' THE XA$ ARRAY.
9540 '      INPUT: XJ,XK,XL SETUP FROM SEARCH MODULE
9550 '      OUTPUT: ENTRY DELETED
9560 '*****************************************************
9570 XK=XA%(XL)
9580 XA%(XJ)=XK
9590 XA%(XL)=-2
9600 XA$(XL)="*"
9610 RETURN
```

Figure 6-16. ADEL Module Listing

Chapter Seven
Secondary Sorts and String Modules

In this chapter we're going to look at another aspect of data storage in the GPM modules, secondary sorts. The GPM are set up to perform ordering of data as it is entered. The ASRCH and AADD modules can be used to order each entry in XA$ as it is entered from the keyboard, disk file, or tape. Order is based on the alphanumeric string in each entry starting with the first field. The SECSRT module provides a "secondary" sort that will reorder the entries in XA$ based upon any field desired. For example, if there are 8 fields in a mailing list, SECSRT will reorder XA$ based on the "zip code" field.

The second topic we'll be talking about here are some of the other modules related to data storage in the GPM. FINDN finds the *n*th entry in XA$. There are also some modules that will "pack" and "unpack" entries to and from their respective fields.

Primary and Secondary Sorts

When the GPM modules were designed, they were made to order data based on the first "field" of each entry. You'll recall that each entry in XA$ can be ordered into any number of fields, each field being separated by an exclamation point character (!). The idea here was that the first field could always be the "key" field — the one that contained the last name for a mail list, the account number for billing, or the part number for inventory. Entries in XA$ could then be ordered on this key.

At times, though, it's nice to be able to reorder the entries based upon some other field, such as zip code, state, or type of account. The SECSRT module allows you to do this.

Because the XA% array is "dedicated" to keeping track of the order based on a normal entry (field 1,2,3 . . .) in XA$, a second array must be used. The second array is designated XB% and is also an integer array of the same size of XA% (the number of entries equals XQ). (See Figure 7-1.)

The second field for the sort could have been specified while the XA$ array was empty and secondary sorting **could** have been done at the same time as primary sorting. However, this would slow down the processing for adding entries by at least a factor of two. It would also prohibit resorting on **another** field, if that were desired.

As a result, SECSRT sorts the entries in XA$ in one huge sort operation at any time the user specifies. After the sort, if any adds, deletes, or modifies are done to XA$, a flag is set that says that the secondary sort is "invalid" and must be redone. Secondary sorts, then, are meant to be used **just prior** to using the data in XA$, as further activity will invalidate the sort.

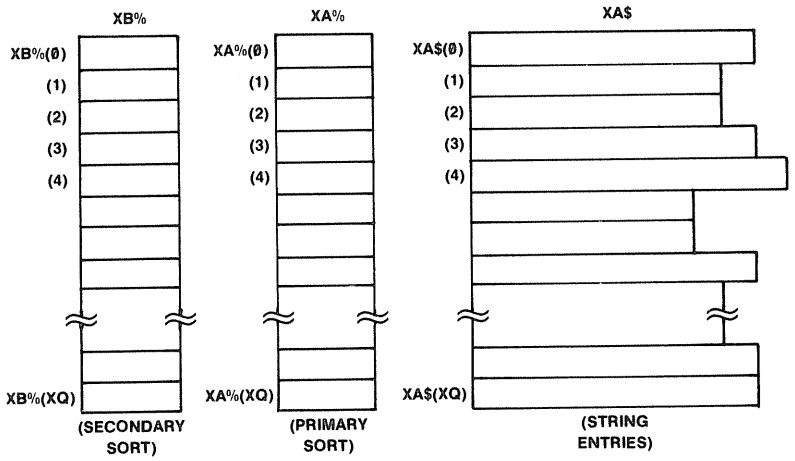


Figure 7-1. XB% Array

The advantages of being able to sort at any time on a second field are obvious. What are the disadvantages? The primary disadvantage is that SECSRT is slow! Expect sorts for 100 entries to be on the order of 1 hour, and sorts for 200 entries to be on the order of 4 hours! (Sorts for 1000 entries take approximately 8 months . . .) The sort time increases drastically as the number of entries increases. With this proviso on SECSRT use, let's look at the operation of SECSRT.

SECSRT Module Operation

The SECSRT module is shown in Figure 7-2. Entry is made to SECSRT with variable YT containing the number of the secondary field for the sort. RETURN is made after the sort is done with XB% containing the sorted entry numbers. A sample call is shown in Figure 7-3.

The first action that SECSRT takes is to initialize the XB% array. A -2 for an unused entry is stored in each entry position of XB%, and then a -1 for "next entry number" is stored in XB%(0).

Next, a check is made of XA%. If it is empty (XA%(0)=-1), no sort is necessary and a RETURN is made.

SECSRT sorts by following the threaded list of XA%. Variable XG holds the "current" entry number from the XA% array, starting with XA%(0).

The code from line 6180 through 6270 takes the current entry and finds the field specified in YT. The double-indented code at 6220 through 6250 finds the start of the specified field by calls to SSRCH (line 1000). After the start of the field is found another call to SSRCH finds the end of the field. XD$ is finally loaded with the characters from the field specified in YT.

```
6000 GOTO 6100 'SECSRT
6010 '*****************************************************
6020 ' THIS IS THE SECONDARY SORT MODULE. IT IS USED TO SORT
6030 ' THE XA$() ARRAY BY A SECONDARY FIELD. THE INDICES TO
6040 ' THE SORT ARE HELD IN THE SECONDARY INDEX ARRAY XB%()
6050 '      INPUT: XA$() AND XA%() ARRAYS
6060 '             YT=# OF SECONDARY FIELD
6070 '      OUTPUT:SORTED ARRAY OF INDICES IN XB%() ARRAY
6080 '*****************************************************
6090 'FIRST INITIALIZE XB%() ARRAY
6100    FOR XI=1 TO XQ-3 STEP 4
6110    XB%(XI)=-2:XB%(XI+1)=-2:XB%(XI+2)=-2:XB%(XI+3)=-2
6120    PRINT @ YL,XI;
6130    NEXT XI
6140 XB%(0)=-1
6150 IF XA%(0)=-1 GOTO 6350
6160 XK=XA%(0)
6170 'UNPACK EACH XA$ ENTRY
6180    PRINT @ YL,XK;"    ";
6190    XZ$=XA$(XK)
6200    XW$="!"
6210    IF YT=1 GOTO 6260
6220       FOR ZI=1 TO YT-1
6230       GOSUB 1000
6240       XZ$=MID$(XZ$,XW+1,LEN(XZ$)-XW)
6250       NEXT ZI
6260    GOSUB 1000
6270    XD$=LEFT$(XZ$,XW-1)
6280    'SEARCH FOR ADD POINT
6290    GOSUB 6360
6300    XB%(XK)=XL
6310    XB%(XJ)=XK
6320    'NOW GET NEXT ENTRY
6330    XK=XA%(XK)
6340    IF XK<>-1 GOTO 6180
6350 RETURN
6360 IF XB%(0)<>-1 GOTO 6390
6370 XJ=0:XL=-1
6380 GOTO 6450
6390 XJ=0
6400 XL=XB%(0)
6410    XY$=XA$(XL):GOSUB 2500
6420    IF XD$<ZW$(YT) GOTO 6450
6430    XJ=XL:XL=XB%(XL)
6440    IF XL<>-1 GOTO 6410
6450 RETURN
```

Figure 7-2. SECSRT Module Listing

```
23580 YT=VAL(ZW$(1))
23590 IF YT<1 OR YT>8 GOTO 23550
23600 'NOW SORT
23610 GOSUB 6000
```

GET FIELD # FOR SORT

SORT — ALL DATA IN XA$ AND XA%

Figure 7-3. SECSRT Module Call

At this point the field from the current entry of XA$ is in XD$. A search must now be made through all of the entries in XA$ to find the "insertion point." Subroutine 6360 accomplishes this. It returns variables XJ, XK, and XL just as ASRCH does in specifying the insertion point. XK is the first "unused" entry number, XJ is the "previous" entry number, and XL is the "next" entry number. These entry numbers will be used to insert the link in XB%; the entry in XA$ will remain undisturbed.

After the insertion point is found, XB%(XK)=XL sets the unused entry in XB% to the next entry number. The last entry number is set to the unused entry number by XB%(XJ)=XK. The next entry number is then found by XK=XA%(XK). This picks up the next entry number from the current XA% entry. If it is not -1, the code loops back to 6180 for processing of the next entry. Each entry number (XK) is printed at the "activity area" to indicate processing.

Subroutine 6360 acts similarly to the ASRCH module. It searches the entries in XA$ to find an insertion point for the field in XD$. Instead of changing pointers in XA%, it uses the secondary sort array XB%.

If there are no entries in XB%, XJ (previous) is set to 0 and XL (next) is set to -1. A RETURN is then made for this first entry.

If there are entries in XB%, the subroutine first finds the first "unused" entry number in XB%. Since XB% was initialized at the beginning of the SECSRT module, this will be the next sequential location in XB%. This value is held in XK.

Next, the entry in XA$ corresponding to each entry in XB% is "unpacked" by calling the SUNPK module (line 2500). This puts all of the fields for the entry into the ZW$ array. The proper field (ZW$(YT)) is compared to XD$. When XD$ is less than the field of the current XA$ entry, a RETURN is made with XJ, XK, and XL set to the proper insertion point.

A complete example of SECSRT action is shown in Figure 7-4.

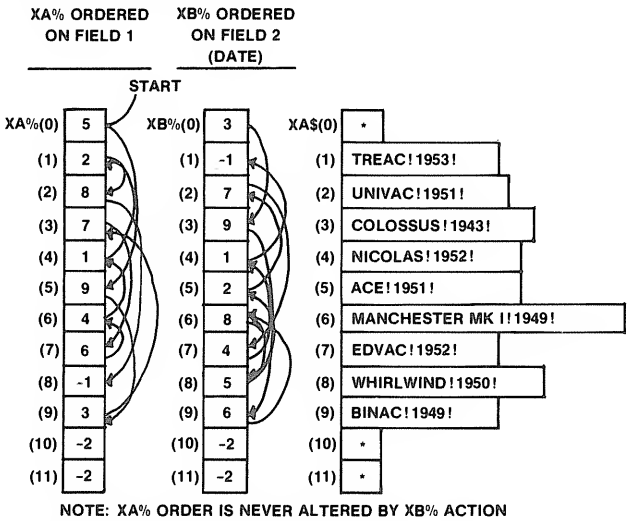


Figure 7-4. SECSRT Operation

FINDN Module Operation

The FINDN module is shown in figure 7-5. It is used to find the *n*th entry in the XA$ array. This is useful because sometimes the number of the entry will be known in place of the "name" or key of the entry.

```
5500 GOTO5650 'FINDN
5510 '************************************************************
5520 ' THIS IS THE "FIND NTH ENTRY" MODULE. IT SEARCHES THE
5530 ' XA% ARRAY TO FIND EITHER A GIVEN ENTRY #, OR TO FIND
5540 ' THE NEXT ENTRY.
5550 '      INPUT: XU=CURRENT # FROM PREVIOUS FIND NTH
5560 '             XS=# TO FIND, 1 TO N OR -1 IF FIND ALL
5570 '             XT=0 IF FIND NTH, 1 IF FIND NEXT
5580 '             YS=0 IF PRIMARY, 1 IF SECONDARY
5590 '      OUTPUT:XM=0 IF ENTRY NOT FOUND ON FIND NTH, 1 IF
5600 '                  FOUND
5610 '             XJ=INDEX TO LAST ENTRY
5620 '             XL=INDEX TO NTH ENTRY OR NEXT ENTRY
5630 '                OR -1 IF NOT FOUND
5640 '************************************************************
5650 IF XU<>0 GOTO 5740
5660 IF XA%(0)=-1 GOTO 5770
5670 XU=1:XJ=0
5680 IF YS=0 THEN XL=XA%(0) ELSE XL=XB%(0)
5690    IF XU<>XS GOTO 5720
5700    XM=1
5710    GOTO 5780
5720    IF XT=1 GOTO 5770
5730    XU=XU+1
5740    XJ=XL:IF YS=0 THEN XL=XA%(XL) ELSE XL=XB%(XL)
5750    IF XL=-1 GOTO 5770
5760    GOTO 5690
5770 XM=0
5780 PRINT @ YL, XU;"    ";
5790 XU=XU+1
5800 RETURN
```

Figure 7-5. FINDN Module Listing

FINDN operates in two modes. The first mode finds the *n*th entry. A sample call (Figure 7-6) would specify that the 50th entry in XA$ is to be found. The second mode finds the "next" entry. The second mode is usually used after finding the first. The 50th entry is found, for example, and then the module is repeatedly called for the 51st, the 52nd, and so forth (Figure 7-7). The second mode is much faster than the first, as it does not start from the beginning of the list.

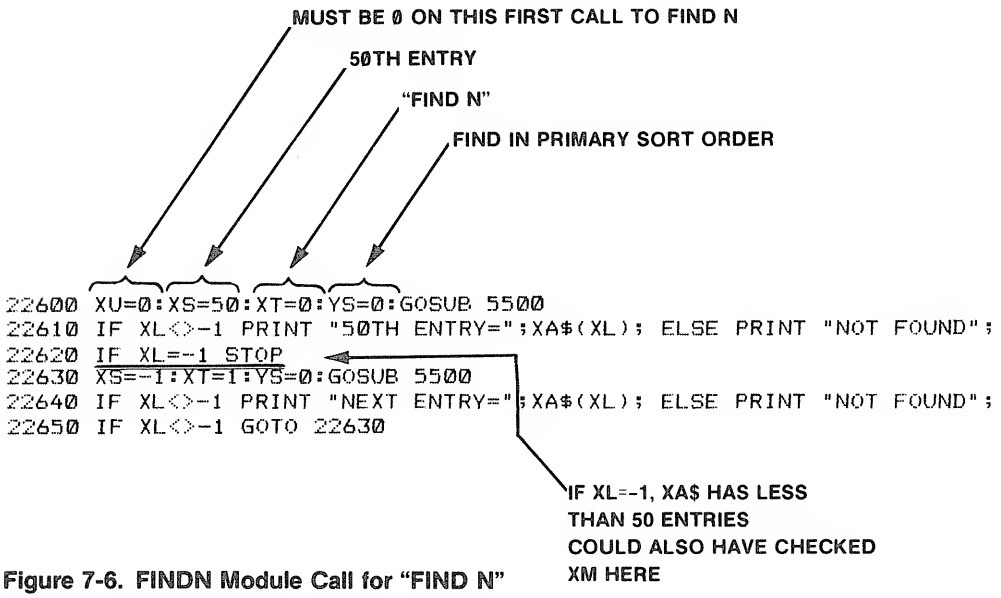


Figure 7-6. FINDN Module Call for "FIND N"

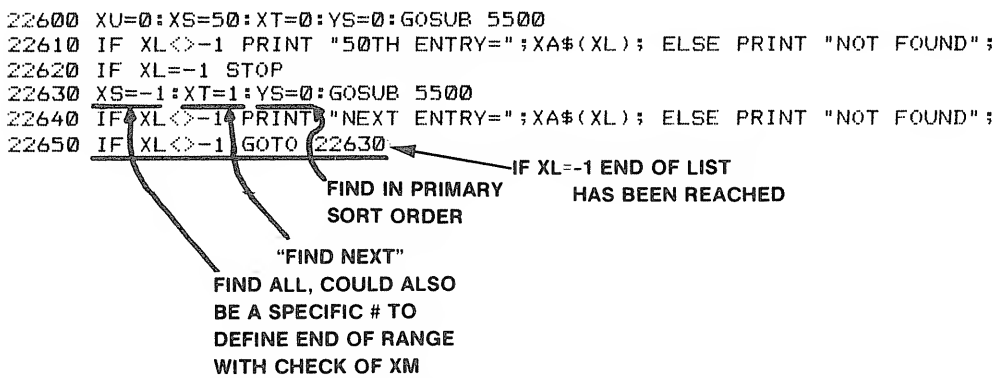


Figure 7-7. FINDN Module Call for "Find Next"

FINDN finds entries from either the primary or secondary array.

Find Nth Entry

In this mode, XU must be set to 0. Variable XS contains the number of the entry

to be found. Variable XT is set to 0 to specify the "find *n*th" mode. Variable YS specifies the *n*th entry of either the primary or secondary array (XA% or XB%).

As XU=0, line 5660 is executed. A check is made of XA%(0). If the entry number is -1, then there are no entries in XA$ and a RETURN is made with variable XM set to "not found."

If there are entries in XA$, XU is set to 1 and XJ to 0. XU holds the "current number" as entries are found. XJ holds the "last" entry number. XL is then set to the first entry number of either the XA% or XB% array.

The code from line 5690 through 5760 is the main loop of FINDN. It goes down through the list of XA% or XB% and counts the number of the entry by adding one to XU. When XU equals XS, the number to find, a RETURN is made with XJ and XL set to the "last" entry number and desired entry number, respectively.

It is possible that the number will not be found if the "number to find" in XS is greater than the number of entries in XA$. In this case a -1 is detected, and a RETURN is made with XM set to 0 for "not found."

The current entry number is displayed in the "activity area" before the RETURN is made.

Find Next Entry

This mode is very similar to the first, except that the count is not made from the beginning of XA% or XB%, but from the last entry number. XU in this case holds the number from a previous call. Variable XS holds a number to find, if a limit is required, or -1 if all entries are to be found. Variable XT holds a 1 for "find next." Variable YS contains a flag for either the primary or secondary array XA% or XB%.

The count is resumed from entry XU and proceeds as in the mode above. A check is made of XT at line 5720, and a RETURN made with XJ and XL set up as before. XM is set if the current entry number corresponds to the value in XS.

SSRCH, SUNPK, AND SPACK MODULES

These three modules are used to "pack" and "unpack" fields and entries in XA$.

SSRCH is used to search a string for a given smaller string - to find USA in SOUSA, for example. It is a general-purpose routine that can be used for any search of this type.

SUNPK takes an entry with fields "delimited" by exclamation points and unpacks the entry into the ZW$ array. It's used to take an entry from the XA$ array and separate into fields for display or comparison.

SPACK is used to do the inverse of SUNPK. It takes fields in the ZW$ array and packs them into XA$ format with exclamation point delimiters (separators).

SSRCH Module Operation

The SSRCH module is shown in Figure 7-8 and a sample call in Figure 7-9. Input to SSRCH is the string to be found, XW$, and the string that may contain XW$, string XZ$. If the XW$ string is found within XZ$, variable XW is returned with the starting "index," or position number, of the string. The position number is a value of 1, 2, 3 and so forth, that corresponds with the character position of the string.

```
1000 GOTO 1090 'SSRCH
1010 '*****************************************************
1020 ' THIS IS THE SEARCH STRING MODULE. IT SEARCHES THE XZ$
1030 ' STRING FOR A SEARCH STRING OF XW$. XZ$ MUST BE LARGER
1040 ' OR EQUAL TO LENGTH OF XW$.
1050 '      INPUT: XW$=STRING TO BE FOUND
1060 '             XZ$=STRING TO BE SEARCHED
1070 '     OUTPUT:XW=START INDEX OF STRING OR -1 IF NOT FOUND
1080 '*****************************************************
1090 IF LEN(XW$)>LEN(XZ$) THEN STOP
1100 XI=LEN(XZ$): XH=LEN(XW$)
1110 IF XW$="!" GOTO 1150
1120    FOR XW=1 TO XI-XH+1:IF MID$(XZ$,XW,XH)=XW$ GOTO 1180
1130    NEXT XW
1140    GOTO 1170
1150    FOR XW=1 TO XI:IF MID$(XZ$,XW,1)="!" GOTO 1180
1160    NEXT XW
1170 XW=-1
1180 RETURN
```

Figure 7-8. SSRCH Module Listing

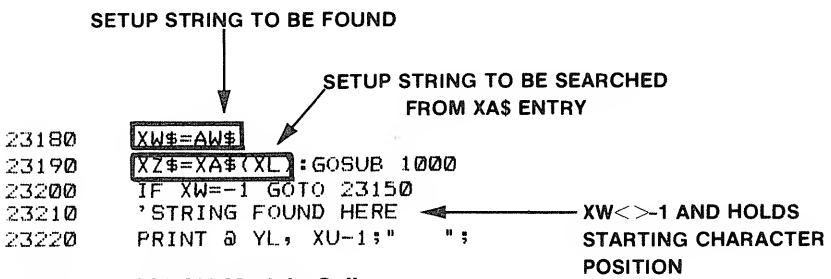


Figure 7-9. SSRCH Module Call

First a comparison is made of the length of the two strings. If the string that is sought is longer than the string that may contain it, the program STOPs.

Variables XI and XH are then set to the length of XZ$ and XW$, respectively.

If the search string is "!", a special fast search is done in lines 1150 through 1160. If the search string is not "!", the code from lines 1120 through 1140 is performed. In either case, if the string is found within XZ$, line 1180 is executed without variable XW being set to -1. If the string is not found, the program "falls through" the loop to set XW equal to -1. Each loop tests for XW$ by "sliding" XW$ along the length of XZ$ for comparison, as shown in Figure 7-10.

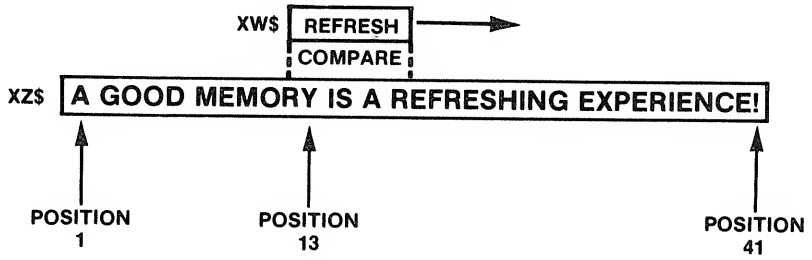


Figure 7-10. SSRCH Module Action

SUNPK Module Operation

This module is shown in Figure 7-11. It calls the SSRCH module to separate the XY$ string into fields which are placed into array ZW$. ZW$(1) corresponds to field 1, ZW$(2) corresponds to field 2, and so forth. A sample call is shown in Figure 7-12.

```
2500 GOTO 2600 'SUNPK
2510 '*************************************************************
2520 ' THIS IS THE UNPACK STRING MODULE. IT FINDS THE INDIV-
2530 ' IDUAL FIELDS OF A STRING CREATED BY THE PACK STRING
2540 ' MODULE BY LOOKING FOR AN EXCLAMATION MARK CHARACTER.
2550 '      INPUT: XY$=STRING TO UNPACK
2560 '      OUTPUT:STRING XY$ IS UNPACKED INTO ZW$(1)-ZW$(N)
2570 '             AND LENGTH OF EACH PUT INTO ZX(1)-ZX(N)
2580 '             NUMBER OF FIELDS IS PUT INTO ZQ.
2590 '*************************************************************
2600 ZQ=1:XW$="!":XZ$=XY$:ZI=LEN(XZ$)
2610    GOSUB 1000 :IF XW=-1 GOTO 2690
2620    ZW$(ZQ)=MID$(XZ$,1,XW-1)
2630    ZX(ZQ)=XW-1
2640    ZI=ZI-XW
2650    IF ZI<1 GOTO 2690
2660    XZ$=MID$(XZ$,XW+1,ZI)
2670    ZQ=ZQ+1
2680    GOTO 2610
2690 RETURN
```

Figure 7-11. SUNPK Module Listing

```
20500 XY$=XA$(XK)
20510 GOSUB 2500
20520 PRINT "FIELD #=";ZW$(Z);" AND LENGTH=";ZX(Z)
```

Figure 7-12. SUNPK Module Call

First of all ZQ is set to 1, XW$ is set to the search string "!", XZ$ is set to XY$ for the search, and ZI is set to the length of XZ$. Variable ZQ is incremented by one through the loop of lines 2610 through 2680. The number of loops will correspond to the fields involved.

Each time through the loop, a call is made to SSRCH (line 1000). A search will be made for "!" in SSRCH. If XW=-1 on RETURN, a RETURN is made from SUNPK.

If XW does not equal -1 on RETURN from SSRCH, then it points to the next position of "!" in the XZ$ string (XY$ string). The leftmost portion of XZ$ is then cut off and stored in ZW$(ZQ) by the MID$(XZ$,1,XW-1). The length of this portion is represented by XW-1. The length is put into ZX(ZQ). Variable ZI is then adjusted to the length of the rightmost portion of the XZ$ string (ZI-XW). If the rightmost position of the string is 0, the unpacking operation is done. Otherwise, XZ$ is set equal to the rightmost portion of the string by MID$(XZ$,XW+1,ZI), ZQ is incremented by one, and the next search is made. A sample operation is shown in Figure 7-13.

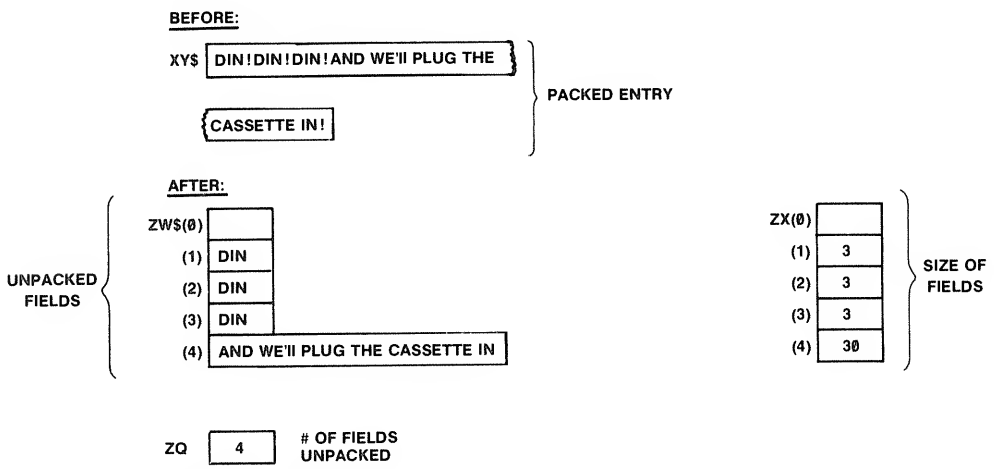


Figure 7-13. SUNPK Operation

SPACK Operation

The SPACK module is shown in Figure 7-14. Its operation is the inverse of SUNPK. Variable ZQ contains the number of fields in the ZW$ array. String variable XY$ is initially null. Each of the fields in the ZW$ array is "concatenated" with XY$ by XY$=XY$+ZW$(XI). A "!" is also added after each field. A sample call and operation are shown in Figure 7-15.

```
6500 GOTO 6600 'SPACK
6510 '*************************************************************
6520 ' THIS IS THE PACK STRING MODULE. IT CONSTRUCTS A STRING
6530 ' OF XY$ MADE UP OF THE "ZP$" FIELDS IN SEQUENCE.
6540 ' THE "DELIMITER" BETWEEN FIELDS IN THE XY$ STRING IS AN
6550 ' EXCLAMATION MARK CHARACTER.
6560 '      INPUT: ZQ=# OF ZP$ FIELDS
6570 '             ZW$(1)-ZW$(N)=FIELDS
6580 '      OUTPUT:XY$ STRING MADE UP OF FIELDS
6590 '*************************************************************
6600 XY$=""
6610    FOR XI=1 TO ZQ
6620    XY$=XY$+ZW$(XI)+"!"
6630    NEXT XI
6640 RETURN
```

Figure 7-14. SPACK Module Listing

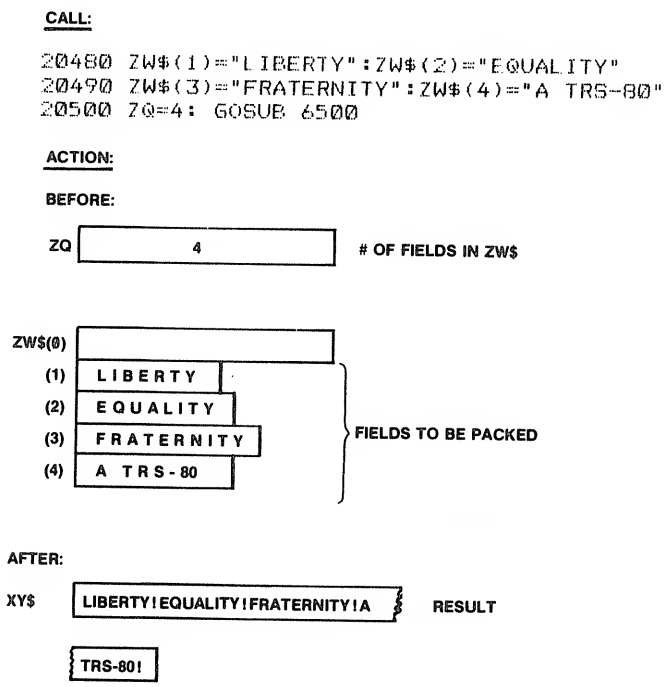


Figure 7-15. SPACK Module Call and Action

Chapter Eight
Line Printer, Cassette, and Disk Operations

We've included some "input/output" modules in the General Purpose Modules to take care of printing operations on the system line printer, and "file" operations on cassette (Models I and III) and disk. (There are other input/output operations possible with the TRS-80s, but we'll leave such things as the down range satellite communications functions for the thirty-second book in this series . . .)

The line printer modules are LPDRIV and REPORT. LPDRIV automatically performs page formatting and report titling. REPORT can be used to "format" any type of printed report by skipping lines, tabbing, printing fields of data, and printing the contents of a variable "counter."

The cassette/disk file modules are CDLOAD and CDSAVE. CDSAVE creates a cassette or disk file from the XA$ array in memory, writing out the entire array in sequential ASCII file format. CDLOAD loads in a previously written file; the load is either an "initialization" type load, or a "merge" of the file data with existing entries in XA$.

Another GPM module discussed in this section is the ERROR module, which is used primarily for disk errors such as "file not found" or "disk full."

Line Printer Operations

The two BASIC commands that can be used for the system line printer are LLIST and LPRINT. We'll only be discussing LPRINT in this section, as LLIST is only used for listing BASIC programs and is not used during execution of applications programs.

LPRINT is similar to PRINT for display work — deceptively simple. It allows you to print a line or partial line of data items, strings, or combination of the two, but does not provide automatic "page formatting" (positioning on the page), just as PRINT does not allow you to position characters on the screen.

LPDRIV Module

The LPDRIV module performs three functions in the GPM:

- Printing a line
- Automatic page formatting
- Automatic titling

The applications program using LPDRIV must first initialize two variables that will probably remain in force throughout the entire application.

ZL is the length of the printer page in lines. The standard number of lines per vertical inch is 6. As most printing will be done on 11 inch long pages, ZL is usually 6 times 11 or 66. If you are using other length pages or special forms, simply set ZL to the lines per page by multiplying lines per vertical inch by length of the page in inches.

ZM is the number of lines per page on the "print image." This figure is initialized the same way — the number of vertical lines per inch is multiplied by the length of the print area, as shown in Figure 8-1.

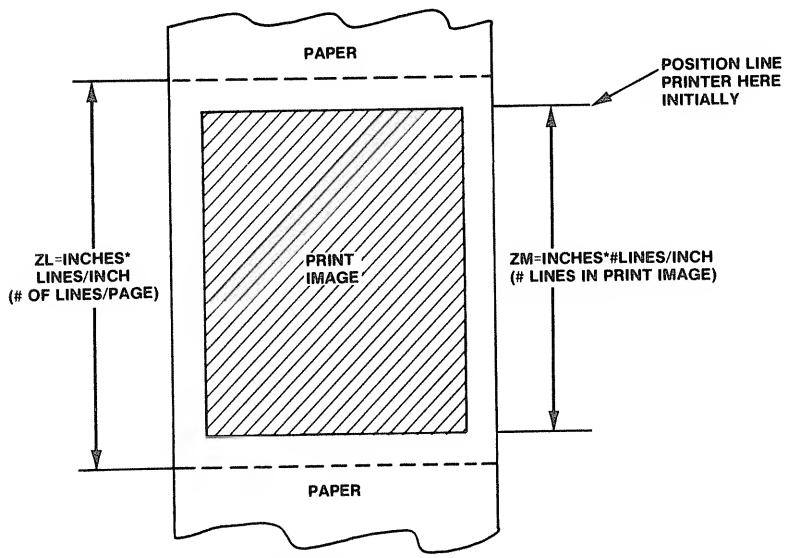


Figure 8-1. Line Printer Variables

LPDRIV Module Operation

The LPDRIV module is shown in Figure 8-2. Entry is made with ZM and ZL set up as discussed. String variable ZM$ is the string to be printed on the current line. ZN$ is a title message that will be printed on every new page. If no title message is to be printed, then ZN$="", a "null" string. A typical call and action for LPDRIV is shown in Figure 8-3.

```
1500 GOTO 1630 'LPDRIV
1510 '*************************************************************
1520 ' THIS IS LINE PRINTER DRIVER. IT OUTPUTS A SINGLE LINE
1530 ' TO THE LINE PRINTER, ADDS ONE TO THE LINE COUNT, AND
1540 ' TESTS TO SEE IF LAST LINE ON PAGE HAS BEEN REACHED. IF
1550 ' LAST LINE HAS BEEN REACHED, A "FORM FEED" IS DONE.
1560 '      INPUT: ZM=NUMBER OF LINES PER PAGE
1570 '             ZL=LENGTH OF PAGE IN LINES
1580 '             ZM$=STRING TO BE PRINTED. IF NULL ("") THEN
1590 '                 "FORM FEED" IS DONE
1600 '             ZN$=PAGE TITLE MESSAGE OR "" IF NO TITLE
1610 '     OUTPUT:LINE IS PRINTED ON SYSTEM LINE PRINTER
1620 '*************************************************************
1630 IF ZJ<>0 GOTO 1660
1640 ZJ=1
1650 ZK=0
1660 IF ZM$="" GOTO 1700
1670 LPRINT ZM$
1680 ZK=ZK+1
1690 IF ZK<>ZM GOTO 1760
1700    FOR ZI=1 TO ZL-ZK
1710    LPRINT"
1720    NEXT ZI
1730 ZK=0
1740 IF ZN$="" GOTO 1760
1750 ZM$=ZN$: GOTO 1500
1760 RETURN
```

Figure 8-2. LPDRIV Module Listing

CALL:

```
20100 XM=50
20110 XL=66
20120 ZN$="SAMPLE PAGE TITLE"
20140 '
20150 '
20160 '
21030 '
22000 ZM$="SAMPLE LINE": GOSUB 1500
```

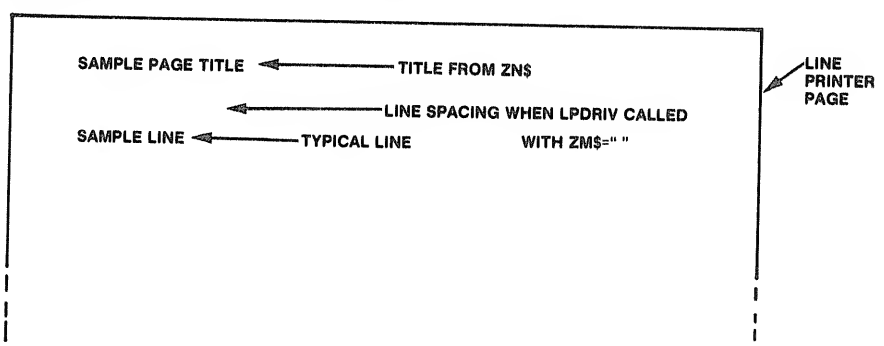


Figure 8-3. LPDRIV Module Call and Action

LPDRIV uses an internal variable ZJ as a "first time flag." ZJ will be zeroed by the CLEAR command in AINIT or by initial loading of the program. If ZJ is zero, this is the first entry to LPDRIV. ZJ is then set to 1, and ZK is set to 0. ZK is the internal variable used to hold the current number of lines that have been printed on the page.

When the application program is first run, the system line printer paper should be positioned to the spot where the first line is to be printed.

Note: Normally LPDRIV initializes a "new page" after receiving a null string. To reset LPDRIV after any repositioning of the line printer paper to the top of the page, a call should be made to LPDRIV with XJ=0.

If ZM$ is equal to a null string (" ") then LPDRIV will automatically perform a "page eject," or new page position. Loop 1700 through 1720 performs this operation by doing an LPRINT" " for a number of times equal to (ZL-ZK). If the length of the page (ZL) is 66 and the current line position (ZK) is line 0, for example, 66 lines will be skipped to position the paper at the top of the next page. If the current line position is 22, 44 lines will be skipped to position the paper at the top of the next page.

If ZM$ does not equal a null string, then it contains a "normal" line to be printed. This is done by LPRINT ZM$. The current line count in ZK is then incremented by one count. Next, ZK is compared to ZM, the number of lines in the print image. If they are equal, a "page eject" must be done, and the loop at 1700 through 1720 is executed. If they are not equal a RETURN is made.

Any time a "page eject" is done, the current line count in ZK is set to 0. At that point, the printer paper is positioned to the first line of the new page. If variable ZN$ is not a null, then a title is to be printed. In this case, ZM$ is set equal to ZN$, and LPDRIV calls itself again! When this action is taken, the title in ZM$ (ZN$) is printed as a normal print line and a RETURN is made after the print. Note that the call back to the beginning is a GOTO rather than a GOSUB. A GOSUB would be invalid, as only one RETURN is eventually made.

LPDRIV With the Model II

The FORMS command of the Model II allows complete page formatting on the Model II system with control of page size, number of lines per page, width, and other functions. You have two options for using LPDRIV with the Model II.

First, you may specify a page length equal to page size in the FORMS command. In this case no "page ejects" will be done by the Model II software, but only by the LPDRIV module. This is the preferred approach.

The second option is to specify a ZM value (number of lines per print image) as a -1 and use the FORMS control. With the second option, however, you will not be able to get automatic page titling, as LPDRIV will never "eject" a page.

REPORT Module

LPDRIV handles the most basic line printer operations — printing a line, top-of-page formatting, and page titling. REPORT expands upon these basic operations.

REPORT uses a list of data items in array XP. The first element of this integer array, XP(0), is the number of items in the array, as shown in Figure 8-4. The item "codes" are as follows:

- 0 = line feed
- -1 = page eject
- -2 = print report counter XN
- -3 to -63 = tab to character position 3 to 63
- 1 to n = print field 1 to n

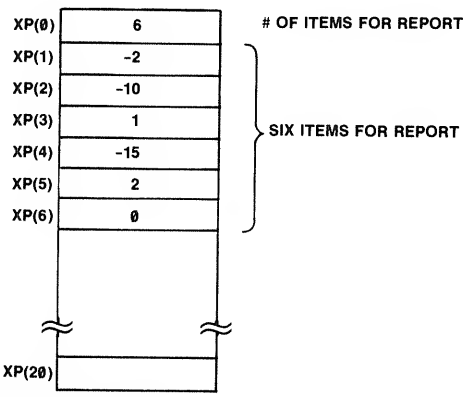


Figure 8-4. REPORT Data Items

REPORT takes each of these items and performs the operation specified. A complete set of operations is defined by the number of items in the XP array. The items in the XP array can be redefined and a new call made to REPORT for another set of operations.

Let's take each of these items and discuss the action of REPORT.

If the item in XP( ) is a -1, a page eject is done to the top of the next page. This code is therefore used to get to the top of the page before printing.

If the item in XP( ) is a 1 to *n*, where *n* is the number of fields used in the XA$ entry, the character string representing the field is printed at the current line printer position. Each field is contained in the ZW$ array in a position corresponding to the field number.

No line feed is done after the field is printed. This code is used to print a field on a line. For example, if a mailing list application was in process, the first and last

names of the addressee might be printed by two consecutive codes of 2,1, which would print the first and last names on the same line.

If the item in XP( ) is a 0, a line feed is performed. This makes the printer skip to the next line. To print a complete mailing list entry with fields as shown in Figure 8-5, for example, the XP array would hold 2,1,0,3,0,4,0,5,6,7,0,0,0, for a six line printout each time REPORT is called.

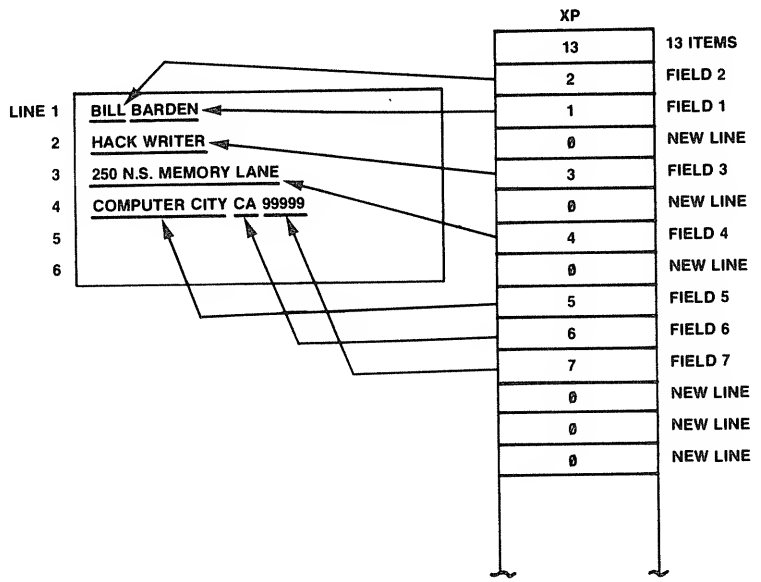


Figure 8-5. REPORT Printing Example 1

If the item in XP( ) is a -3 to -63, then a "tab" operation would be done. This would move the line printer to a character position on the print line corresponding to the tab value. To print a last name, first name at tabs 10 and 30, for example (see Figure 8-6), the XP array would hold -10,2,-30,1,0.

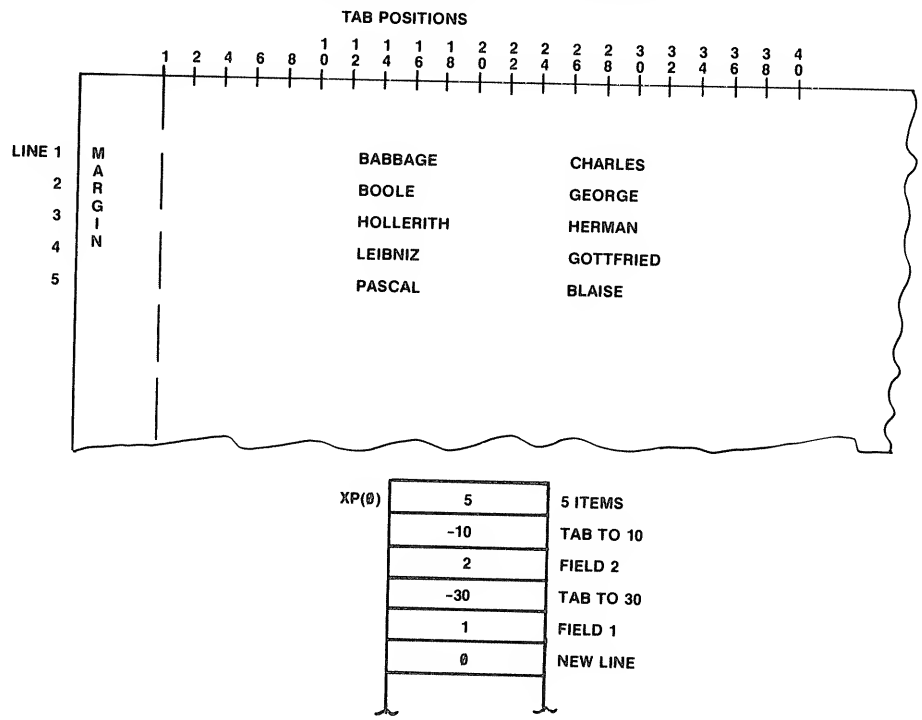


Figure 8-6. REPORT Printing Example 2

If the item in XP( ) is a -2, then the report counter variable XN is printed. XN is automatically incremented by one each time it is printed. If the XP array defined inventory items, for example, each set of inventory fields printed would cause XN to be incremented by 1 if the codes in XP included a "print report counter" code of -2. To print a list of part numbers and descriptions together with an index number, for example, the XP array might contain -2,-5,1,2,0, as shown in Figure 8-7. **XN would be set to 1 prior to the first call to** REPORT**; it would then increment automatically from the initial value of 1.**

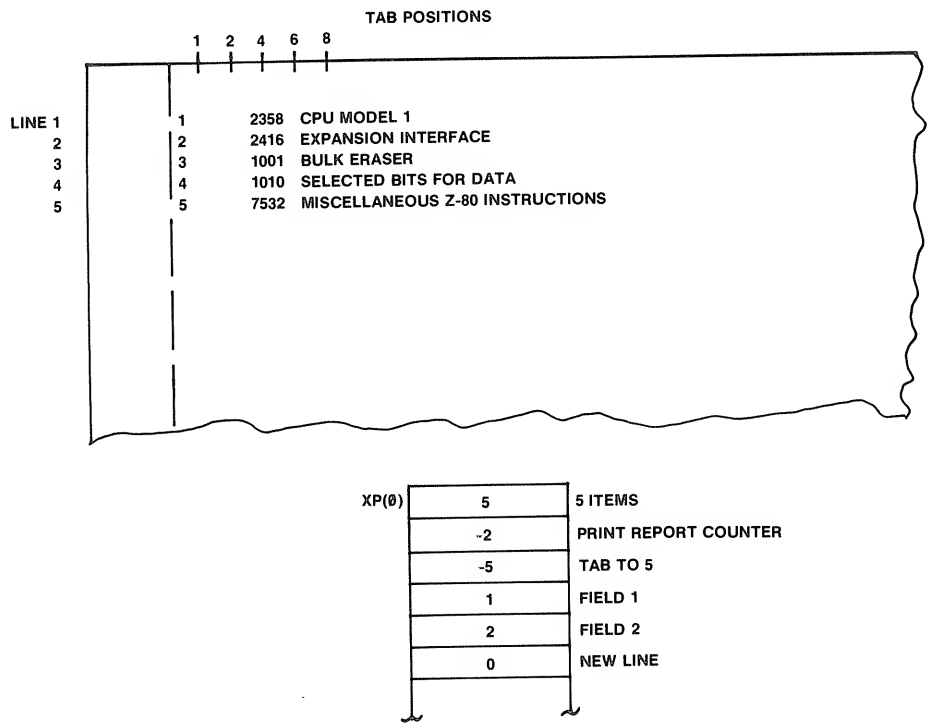


Figure 8-7. REPORT Printing Example 3

A complete example of REPORT is shown in Figure 8-8, which uses REPORT and LPDRIV to print an entire page of information from entries in the XA$ array. Other GPM modules are called to get the XA$ entries (FINDN) and to unpack them into the field array, ZW$ (SUNPK).

```
1       EARL ECHEL 627 FALLINGWATER DR. HUNTINGTON BEACH CA 92647
2       ROBERT EDISO KINROSS RD #5 BROOKLINE MA 02146
3       F. EDWARDS P.O. BOX 9999 SCOTTSDALE AZ 85252
4       CARL ELKINS 9414 SO. FAIRPLAIN AVE. WHITTIER CA 90601
5       EDWARD ERVIN 17243 PURCHE GARDENA CA 90249
6       DENNIS EVANS 8906 AMBER TRAIL SUN CITY AZ 85351
7       JAMES C. FAGAN 326 1/2 W. FOOTHILL BL. CLAREMONT CA 91711
8       DALE FAIRFIELD 5514 ADDINGTON DR. ANAHEIM CA 92807
9       MIKE FANDICH 929 UNIVERSITY DR. COSTA MESA CA 92627
10      JOHN FAULK 3331 E. COMMONWEALTH FULLERTON CA 92631
11      BILL D. FIVES 5444 SILVA LAKEWOOD CA 90713
12      K.K. FOSTER 1234 17TH DRIVE LAGUNA BEACH CA 92651
13      CARL FRANK 8965 PARK MILANO CALABASAS CA 91302
14      JIM FREEMAN 1735 DARYN CANOGA PK. CA 91307
15      TEG FULLER 3888 SOUTH TOWNSEND SANTA ANA CA 92704
16      JIM GABRIEL 1308 WESTMINISTER WESTMINISTER CA 92683
17      P.R. GAGLIANO 8118 SMITHE DR. #8 HUNTINGTON BCH CA 92646
18      JOHN GALLAHER 1388 LINDFORD LANE YORBA LINDA CA 92686
19      JAMES GARNER 8880 SOUTH JASON COURT ENGLEWOOD CO 90110
20      GEORGE GARON 123 W. 123RD #6 ANAHEIM CA 92801
21      JOSEPHINE GARULA 1932 HARDWICK AVE. LAKEWOOD CA 90713
22      MICHAEL GATES 3634 ONATEO TRAIL CHAPEL HILL NC 27514
```

Figure 8-8. REPORT Printing Example 4

REPORT Module Operation

The REPORT module is shown in Figure 8-9.

```
7500 GOTO 7640 'REPORT
7510 '*****************************************************
7520 ' THIS IS THE REPORT MODULE. IT PRINTS A REPORT ON THE
7530 ' SYSTEM LINE PRINTER AS DEFINED BY A LIST OF "ITEMS".
7540 '      INPUT: XP(0)=NUMBER OF ITEMS
7550 '             XP(1)-XP(N)=ITEMS
7560 '             XN=AUTO-INCREMENTING COUNTER
7570 '      OUTPUT:ITEMS DEFINE PRINTING OF FIELD DATA
7580 '      ITEMS: 0=LINE FEED
7590 '             1-N=FIELD N STRING IN ZW$(1)-ZW$(N)
7600 '             -M=TAB TO M
7610 '             -1=PAGE EJECT
7620 '             -2=PRINT REPORT COUNTER XN
7630 '*****************************************************
7640 IF XP(0)<1 THEN STOP
7650 ZM$=""
7660    FOR XI=1 TO XP(0)
7670    IF XP(XI)<>0 GOTO 7700
7680    IF ZM$="" THEN ZM$="          "
7690    GOSUB 1500 :ZM$="":GOTO 7770
7700    IF XP(XI)>0 THEN ZM$=ZM$+ZW$(XP(XI))+" ": GOTO 7770
7710    IF XP(XI)<-2 THEN ZM$=ZM$+STRING$(-XP(XI)-LEN(ZM$)," "):GOTO 7770
7720    IF XP(XI)<>-2 GOTO 7760
7730    ZM$=ZM$+STR$(XN)
7740    XN=XN+1
7750    GOTO 7770
7760    ZM$="":GOSUB 1500
7770    NEXT XI
7780 RETURN
```

Figure 8-9. REPORT Module Listing

The first action REPORT takes is to test XP(0) for no entries (0 value). A STOP is done to inform the user of the error.

Next, ZM$ is set to a null string (" "). Variable ZM will hold the current string to be printed during the course of REPORT.

Lines 7660 through 7770 constitute the loop that takes each item from the XP array and performs the necessary action. The loop is executed a number of times equal to the number of items in XP (FOR XI=1 TO XP(0)).

A check is first made for the item being equal to 0 (line feed). If it is a zero, ZM$ contains a line of data to be printed or may be a null string. If it is a null string, ZM$ is set to a blank line for printing, otherwise no line feed would result. The LPDRIV module (line 1500) is then called for printing the line. After the call to LPDRIV, ZM$ is reset to " " and a GOTO 7770 will increment to the next item in XP.

If the item is not equal to 0, a check is made for an item greater than 0. This would be a field item. If the item is greater than 0, the field item in ZW$(XP(XI)) is **appended to** the "working string" of ZM$ and a GOTO 7770 picks up the next item. Note that at this point nothing has been printed (no call has been made to LPDRIV).

If the item is not 0 or greater than 0, it is a negative item of page eject, print report counter, or tab. If the item is **less than** -2, it is a tab item. In this case the "working string" of ZM$ must be "padded out" with blanks to the tab position specified. The number of blanks required is the tab position minus the current length of the string in ZM$. This is (-XP(XI)-LEN(ZM$)). A STRING$ of blanks is generated and added to ZM$. A `GOTO 7770` picks up the next item. Note again that no line has been printed.

It is possible to specify a tab position less than the current print position, producing an illegal function call error. Always make certain that fields that are printed do not exceed the tab position specified!

If the item is not 0, greater than 0, or less than -2, then it must be -1 (page eject) or -2 (print report counter).

If the item is not equal to -2 it is a page eject. ZM$ is set to " ", which is the signal to `LPDRIV` to eject a page. A call to `LPDRIV` at line 1500 ejects the page and the next item is picked up.

If the item is -2, the report counter XN is converted to a string from a numeric variable by STR$(XN). This string is then appended to ZM$. The count in XN is "bumped" by one, and the next item is picked up by a `GOTO 7770`. Here again, no line is printed.

Cassette Operations

Although provision is made for cassette files in the GPM modules, their use is not recommended. The chief reason for this is inconvenient multiple file operations.

The cassette drive reads or records data at a rate of about 63 or 189 characters per second (Model I and Model III, respectively). However, each **record** may be only 248 characters long, as shown in Figure 8-10. Between each record is an **inter-record gap** of **leader** or zero data. This gap is used to get the cassette tape up to speed before cassette read or write operations take place. 200 entries of 40 characters per entry would take about 15 minutes to write or read on the Model I and 5 minutes on the Model III, which is a tolerable time, even though much slower than the approximately 15-second read/write rate for the same file on disk (these times do not include the BASIC processing for each record, which would add several minutes).

Cassette speeds are therefore marginally acceptable. The main disadvantage of cassette lies in the fact that it is a **sequential-access** device. Both data within a file and files themselves must be accessed sequentially.

First of all it's inconvenient to read in **any** file, because the cassette must be positioned by a REWIND, by manually positioning the tape, or by searching for

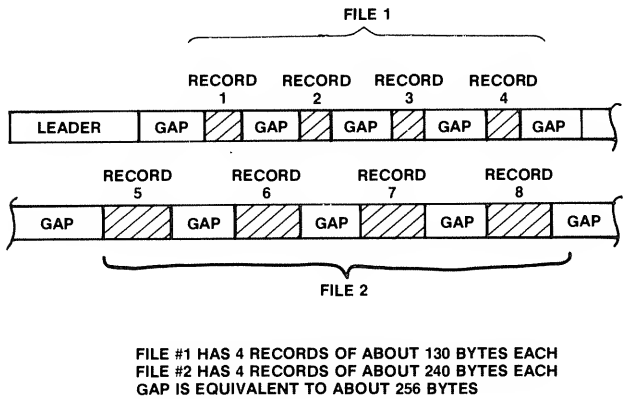


Figure 8-10. Cassette Storage

a given file name. The latter method adds still more time to the operation and is still a manual operation initially to REWIND the tape. (Unfortunately, there is no TRS-80 "automated finger" peripheral device to handle cassette button pushes . . .)

Once the file is found, data within the file must be accessed sequentially. The first records (entries) must be read before a desired record is found. There is no way to read in the 23rd or the 56th record without first going through all of the preceding ones. Figure 8-11 shows what we mean.

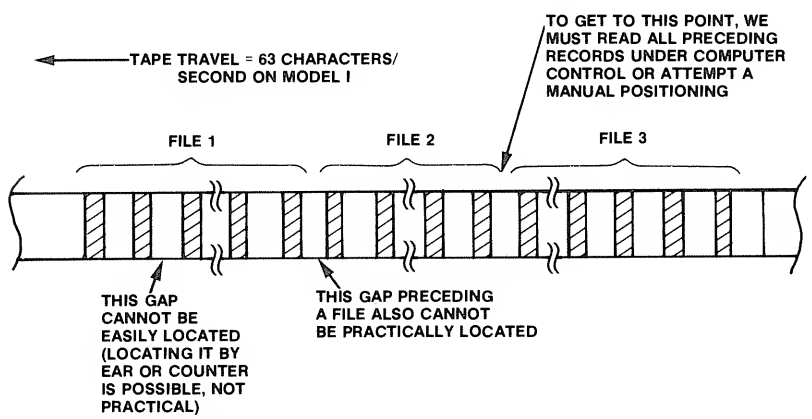


Figure 8-11. Multiple Files with Cassette

This lack of **random-access** capability on cassette for either files or data within a file makes it a poor choice for large applications programs of any type. In the GPM, therefore, we'll be concentrating on disk operations, although we will mention the built-in cassette capability for those of you who don't yet have a disk drive or two. (Any cassette operations you wish to implement can easily be converted to disk at a later time.)

Disk Operations

General Disk Considerations

Before we talk about disk files, let's review some general facts about disk operation in general. Each diskette is a circular piece of mylar with a magnetic coating. Standard Radio Shack diskettes allow writing data on one side only.

Each diskette is divided into a number of **tracks**. A track is a concentric circle around the disk as shown in Figure 8-12. Each track is divided into **sectors.**

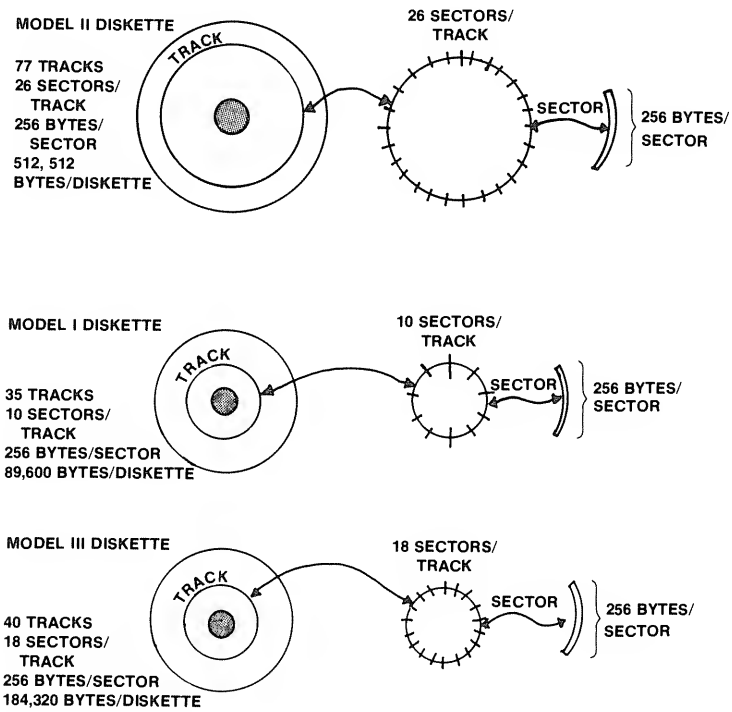


Figure 8-12. Diskette Layout

The TRSDOS Disk Operating System for the Model I, II, or III keeps track of which sectors are **allocated** for disk **files**. A file is any logical grouping of similar **records**.

There are two general types of files in BASIC, **sequential** files and **random** files.

Sequential files are somewhat similar to cassette files. They are long strings of data, as shown in Figure 8-13. The file crosses over sector boundaries, and there is no relationship to the record (entry) number in the file to the sector number on the diskette. TRSDOS records the starting track and sector and the file is read in as a string of data.

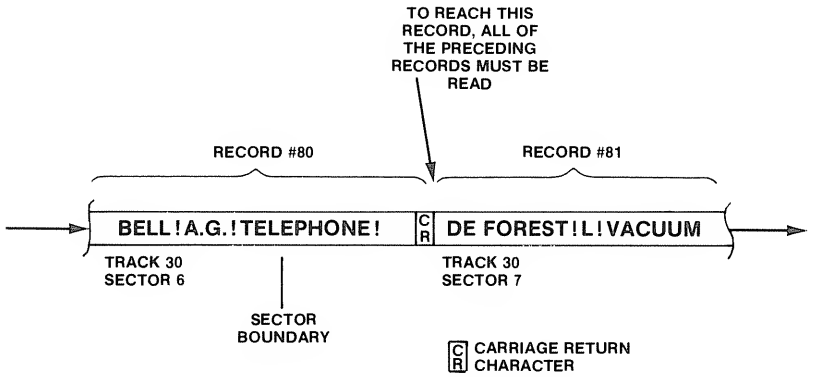


Figure 8-13. Sequential Files on Disk

The user must place some recognizable piece of data as an "end-of-file" marker at the end of the file, or use the EOF (end-of-file) BASIC function to avoid reading in more data than the file contains. If the first approach is used, the mark might be something as innocuous as a character not used in the file, such as an asterisk (*).

Random files are organized in records, as shown in Figure 8-14. Each record is **fixed-length**. Data can be read from the file by GETting a specific record number and can be stored by PUTting a specific record number. As any record can be read or written at any time, access of data within the file is on a "random" rather than sequential basis.

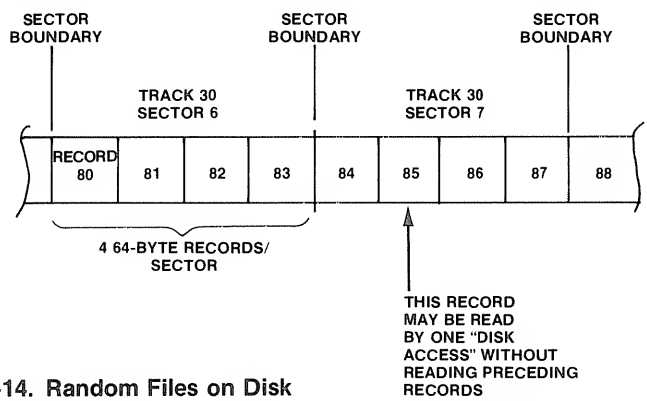


Figure 8-14. Random Files on Disk

There are advantages and disadvantages of each type of file structure. Sequential files offer these advantages:

- Generally they are more efficient in disk storage as a record does not have to be "padded out" to a fixed length
- They are somewhat easier to use in setting up commands
- When data is accessed sequentially, as in reading in a large number of sequential entries, the disk operations are fairly fast

The advantages of random files are:

- Individual records or entries can be easily accessed on disk, allowing such operations as "disk sorting"
- Fields within records can be easily accessed
- Numeric data can be more easily handled and "packed"

GPM Disk Operations

The GPM CDLOAD and CDSAVE modules are geared to sequential files, primarily to make most efficient use of RAM memory. Multiple files may be written or read, and any number of files may be "merged" into one file.

Files are written on disk directly from the XA$ array. The contents of XA$ are written as a sequence of entries, following the order of the XA% "pointers." The last record of the file is an "*", marking the "end-of-file." An example of this approach for a small number of entries in XA$ is shown in Figure 8-15.

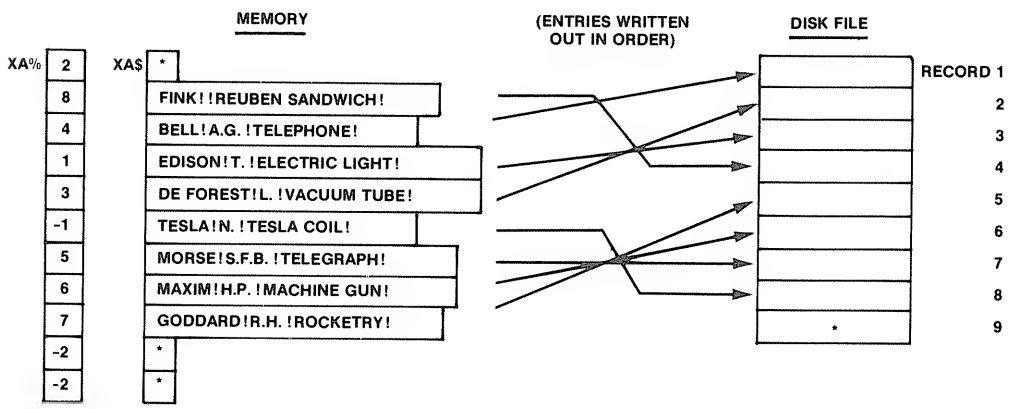


Figure 8-15. Disk Operations in the GPM

All GPM files are in ASCII format. This means that the data appears on disk as a string of ASCII characters. Because the files are in this format they can be LISTed on the screen by the TRSDOS LIST command, PRINTed on the system line printer by the TRSDOS PRINT command, and read in under SCRIPSIT for editing operations if absolutely necessary.

Using the ASCII format means that you have excellent control over the disk file operations, as files may be examined or corrected if necessary! (This is a boon to paranoids such as the author who occasionally think that computer systems are "out to get them" . . .) Figure 8-16 shows how the file used in Figure 8-15 would appear when PRINTed by the TRSDOS PRINT command.

```
BELL!A.G. !TELEPHONE!
DE FOREST!L. !VACUUM TUBE!
EDISON!T. !ELECTRIC LIGHT!
FINK!!REUBEN SANDWICH!
GODDARD!R.H. !ROCKETRY!
MAXIM!H.P. !MACHINE GUN!
MORSE!S.F.B. !TELEGRAPH!
TESLA!N. !TESLA COIL!
*
```

Figure 8-16. Printing GPM Files

CDSAVE Module Operation

The CDSAVE module is shown in Figure 8-17. It takes the XA$ array and writes it out to disk (or cassette) as a sequential file in ASCII under a user-specified file name. A typical call and action is shown in Figure 8-18.

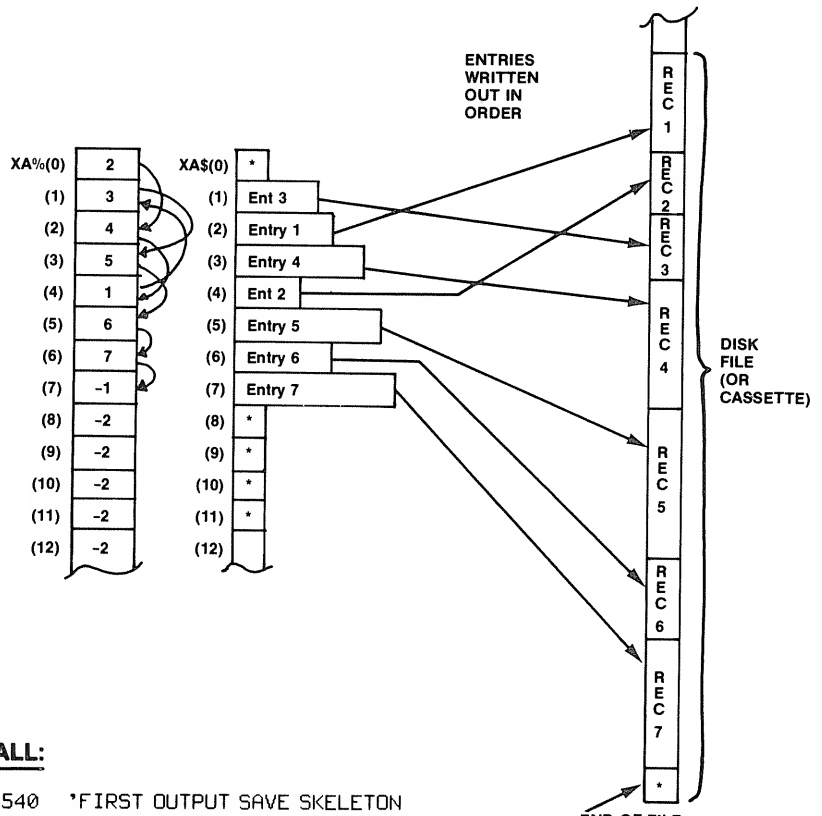
```
12500 GOTO 12580 'CDSAVE
12510 '*****************************************************
12520 ' THIS IS THE CASSETTE/DISK SAVE MODULE. IT SAVES THE XA$
12530 ' ARRAY AS A CASSETTE OR DISK FILE IN GPM FORMAT.
12540 '      INPUT: ZW$(1)="C" FOR CASSETTE OR "D" FOR DISK
12550 '             ZW$(2)=FILENAME (DISK ONLY)
12560 '     OUTPUT:XA$ ARRAY WRITTEN TO CASSETTE OR DISK
12570 '*****************************************************
12580 IF ZW$(1)="C" GOTO 12600
12590 OPEN "O",1,ZW$(2)
12600 XU=0
12610    XS=-1:XT=1:GOSUB 5500
12620    IF XL=-1 GOTO 12650
12630    IF ZW$(1)="D" THEN PRINT #1,XA$(XL) ELSE PRINT #-1,XA$(XL)
12640    GOTO 12610
12650 IF ZW$(1)="D" THEN PRINT #1,"*" ELSE PRINT #-1,"*"
12660 IF ZW$(1)="D" THEN CLOSE 1
12670 RETURN
```

Figure 8-17. CDSAVE Module Listing

CDSAVE is called with ZW$(1) containing a "C" for cassette or a "D" for disk. ZW$(2) contains a disk file name; ZW$(2) is ignored for a cassette file.

If ZW$(1) is "D" for disk, the disk file is first **OPENed** by the OPEN "O",1,ZW$(2) statement. The OPEN operation makes an entry in the TRSDOS directory under the file name in ZW$(2). It essentially tells TRSDOS "here comes a file named xxxxx." The "O" in the OPEN statement indicates that the file will be an **O**utput file.

The "1" indicates that the file will use **buffer** number 1. A buffer is a memory area in which the data is assembled into 256-byte records for output to the disk. Disk writes are done a sector at a time, so the separate pieces of data are stored in the buffer until 256 bytes are available, at which time a disk write of one sector is performed. This eliminates constant writing to add new pieces of data to the



CALL:

```
24540  'FIRST OUTPUT SAVE SKELETON
24550  ZP=55:ZQ=2:ZP$(0)="SAVE FILE"
24560  ZP$(1)="SAVE ON CASSETTE (C) OR DISK (D)?"
24570  ZP$(2)="DISK FILENAME?":ZR(1)=1:ZR(2)=15:GOSUB 8000
24580  'NOW READ IN RESPONSES
24590  GOSUB 8500 : IF XX=1 GOTO 24630
24600  IF ZW$(1)<>"C" AND ZW$(1)<>"D" GOTO 24550
24610  'SAVE TO CASSETTE OR DISK
24620  GOSUB 12500          <-- ZW$(1) - ZW$(2)
24630  RETURN                   HOLD DEVICE, NAME
```

Figure 8-18. CDSAVE Module Call and Action

disk file, which would add several seconds for each piece of data written to the disk.

Next, a call is made to the FINDN module to find the next record from XA$. As XU was set to 0, the first call finds the very first XA$ entry. The entire XA$ entry is output to the disk file (to the disk buffer) by the `PRINT #1,XA$(XL)`. XL contains the entry number of the XA$ entry from the `FINDN` module.

The loop from 12610 through 12640 writes each "next entry" from XA$ until XL=-1, indicating that the last entry in the XA$ array has been reached.

At this point all data has been written to the disk file with the possible exception of the the last partially filled buffer, as shown in Figure 8-19. To write out this buffer, and to properly tell TRSDOS that all data in the file has been output, a CLOSE operation is done. `CLOSE 1` writes the last buffer and initiates TRSDOS actions such as writing an end-of-file mark and completing the directory entry.

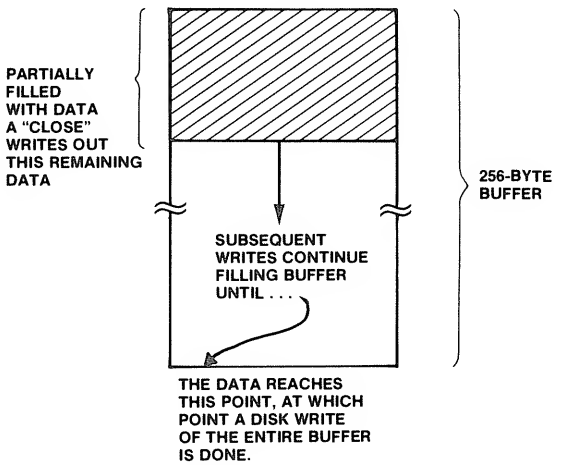


Figure 8-19. CLOSE Action

If ZW$(1) contains a "C" for cassette operations, all of the above actions are performed with the exception of the OPEN and CLOSE. Writes to cassette are done by a `PRINT #-1,XA$(XL)`.

CDLOAD Module Operations

The `CDLOAD` module is shown in Figure 8-20. It loads a previously written GPM file into the XA$ array and sets up the XA% array pointers as it does so. There are two types of loads that may be done by `CDLOAD`.

The first type of load is an "initialize" load. In this case the XA$ array is first "cleared" of all previous entries. The disk file is then read sequentially into XA$(1), XA$(2), XA$(3), and so forth until the last entry has been input. As disk file entries are arranged in order, the initialize load results in an ordered arrangement of the entries in XA$ on completion. This is a "fast" load that is

```
12000 GOTO 12110 'CDLOAD
12010 '*****************************************************
12020 ' THIS IS THE CASSETTE/DISK LOAD MODULE. IT LOADS A
12030 ' CASSETTE OR DISK FILE IN GPM FORMAT INTO THE XA$ ARRAY
12040 ' AND ADJUSTS THE XA% POINTERS. THE LOAD IS EITHER AN
12050 ' INITIALIZE TYPE OR A MERGE.
12060 '     INPUT: ZW$(1)="C" FOR CASSETTE OR "D" FOR "DISK"
12070 '            ZW$(2)=FILENAME (DISK ONLY)
12080 '            ZW$(3)="I" FOR INITIALIZE OR "M" FOR MERGE
12090 '     OUTPUT:FILE READ INTO XA$ AND XA% ADJUSTED
12100 '*****************************************************
12110 IF ZW$(3)="M" GOTO 12200
12120 'INITIALIZE ARRAYS IF "INITIALIZE"
12130    FOR XI=1 TO X0-3 STEP 4
12140    XA%(XI)=-2:XA%(XI+1)=-2:XA%(XI+2)=-2:XA%(XI+3)=-2
12150    XA$(XI)="*":XA$(XI+1)="*":XA$(XI+2)="*":XA$(XI+3)="*"
12160    PRINT @ YL,XI;
12170    NEXT XI
12180 PRINT @ YL, "      ";
12190 XA%(0)=-1:XA$(0)="*"
12200 IF ZW$(1)<>"D" GOTO 12260
12210 ZZ(0)=1: ZZ(1)=53: ZZ$(1)="FILE NOT FOUND"
12220 OPEN "I",1,ZW$(2)
12230 ZZ(0)=0
12240 IF XX=1 GOTO 12390
12250 'NOW READ IN ENTRIES ONE AT A TIME
12260 XI=0
12270    IF ZW$(1)="C" THEN INPUT#-1,XY$ ELSE INPUT #1,XY$
12280    IF XY$="*" THEN GOTO 12380
12290    IF ZW$(3)="M" GOTO 12350
12300    'INITIALIZE TYPE ADD HERE
12310    XA%(XI)=XI+1:XA$(XI+1)=XY$:XI=XI+1:XA%(XI)=-1
12320    PRINT @ YL,XI;
12330    GOTO 12270
12340    'MERGE TYPE ADD HERE
12350    XD$=XY$:GOSUB 5000
12360    GOSUB 7000
12370    GOTO 12270
12380 IF ZW$(1)="D" THEN CLOSE 1
12390 RETURN
```

Figure 8-20. CDLOAD Module Listing

used for initializing a file at the beginning of an application program. Figure 8-21 shows this type of load.

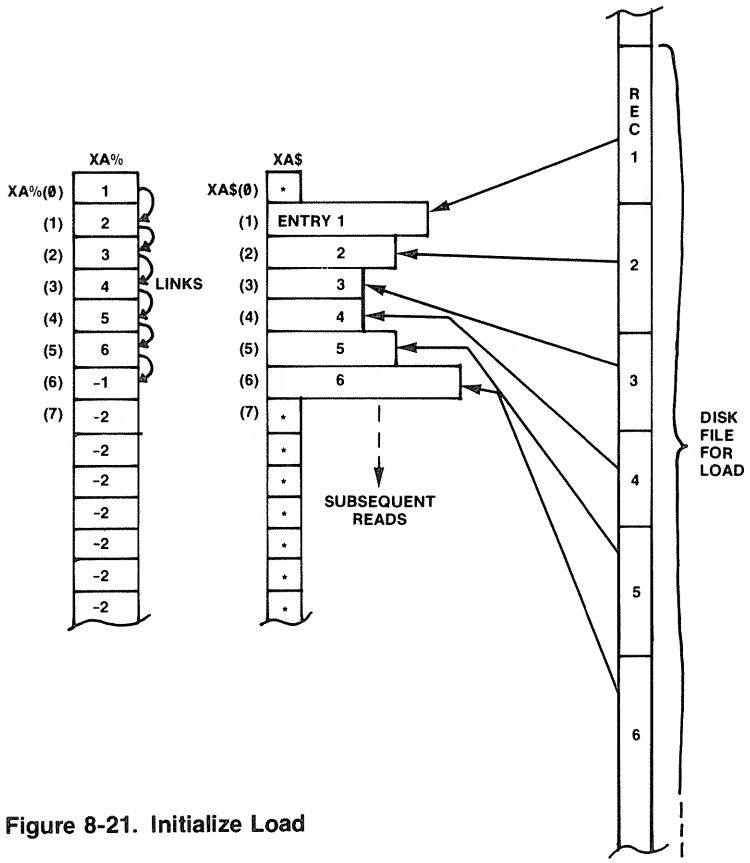


Figure 8-21. Initialize Load

The second type of load is a "merge" load. In this case the entries in XA$ are not cleared, but left intact. Each of the disk file entries are then read, and each entry is added to the XA$ array. This add operation results in each entry being "inserted" at the proper place in XA$ with the pointers in XA% adjusted for the insert.

The merge load allows one or more disk files to be merged together with or without existing data in XA$. Because the add takes some "overhead," this type of load is slower than the initialize. Figure 8-22 shows this type of load for a merge of two disk files.

On entry, ZW$(1) contains a "C" for cassette or a "D" for disk. ZW$(2) contains a disk file name for the file to be used in the load. ZW$(3) contains an "I" for initialize, or an "M" for merge.

If ZW$(3) is an "I" for initialize, the XA$ and XA% arrays are "cleared." The code in lines 12130 through 12180 sets every entry in XA% to -2 (unused) and every entry in XA$ to "*" (unused). The clear operations are done in groups of four entries to reduce the time for clearing. For each set of four clears, the

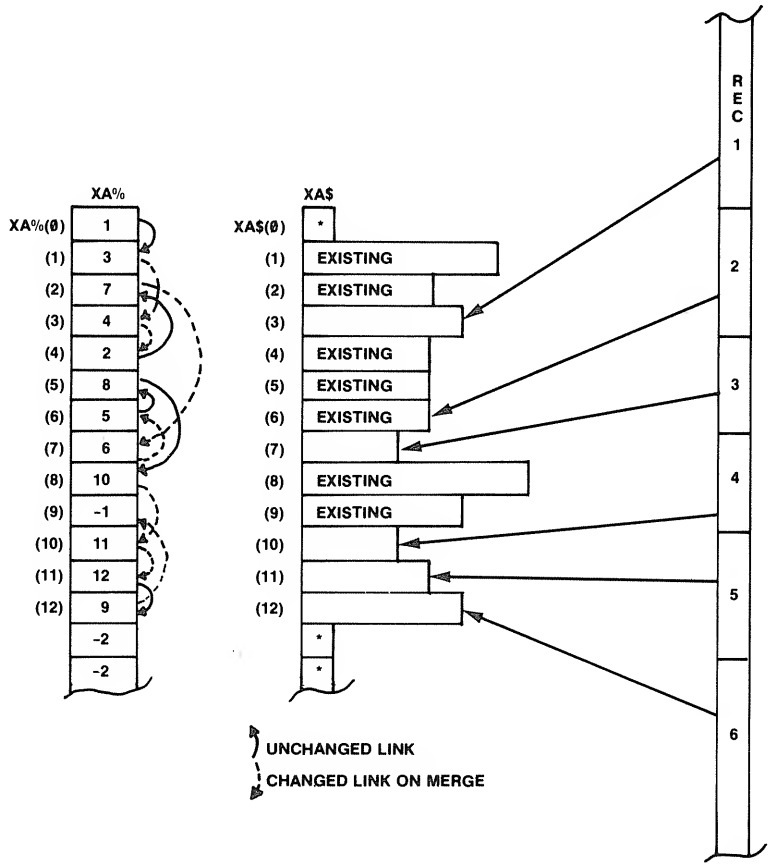


Figure 8-22. Merge Load

current value of XI is printed in the "activity area." The last action in the clear is to set XA%(0) to "no entries" (-1) and XA$(0) to "*".

If a disk file is to be loaded, an OPEN must be done. The OPEN on a disk read serves a similar purpose as an OPEN on a disk write - it causes TRSDOS to find the specified file name and prepare for reading in data from the file. The OPEN specifies that an input operation ("I") will be taking place, that buffer 1 will be used, and that the file name will be found in ZW$(2). Here again a buffer full of data is used. Reads of one sector at a time are done; data is then taken from the buffer one piece at a time for each INPUT.

The action taken at line 12210 is in case of a "non-catastrophic" error. If the wrong file name is specified, it is convenient to have the BASIC program print out FILE NOT FOUND instead of stopping. The ERROR module discussed later in this chapter handles this type of error, and we'll cover the actions in detail there.

If an initialize load is to be done, the loop at 12270 through 12330 is followed. Each time through the loop an INPUT from the disk file (buffer) is done by the INPUT#1,XY$. This reads the next entry into string variable XY$. If this entry is "*" the last entry has been read and line 12380 is executed. If the "*" end-of-file is not present, the following actions are taken:

- XA%(XI) is set to XI+1. This sets the pointer in XA% to the next consecutive entry.
- XA$(XI+1) is set to XY$. This stores the entry just read into the XA$ array.
- XI is incremented by one. This is in lieu of a FOR XI=1 TO . . . loop.
- The next XA% entry is set to -1 to mark the end of the "chain."

For each entry added, the value of XI is displayed in the activity area to reassure the user that something is happening!

If a merge load is to be done, the code from line 12270 through 12370 is executed. First, the entry is read into XY$ as above. In this case, though, XD$ is set equal to XY$ and a call is made to the ASRCH module (line 5000) to find the proper insertion point in XA$. After the point is found, the AADD module (line 7000) is called to add the entry. The loop then repeats until the end-of-file "*" is found.

When the last file entry of "*" is found, both the initialize and merge loads go to line 12380. The CLOSE here closes the disk file operations of buffer 1.

For cassette operations all functions above are identical with the exception of the OPEN and CLOSE. The next entry for cassette is read into string variable XY$ by an INPUT#-1.

Error Operations

There are a number of different types of errors in BASIC programs. Many are simply "logic" errors caused by improper coding of the program. Making a spelling error in a BASIC command might result in a "syntax" error, for example. An "illegal function call" might occur if an invalid array number were used, such as XA$(-1). In these cases the BASIC interpreter stops execution of the program and prints out the type of error and line number. This is well and good, as the user must correct the logic error in the program and can tolerate the termination of the program.

There are other types of errors, however, which are not logic errors, but are "recoverable" errors. In these cases, the BASIC program should not stop, but should continue so that the user can "try again." An example of this type of error is attempting a load of a file with the wrong file name. In this case the user may have simply made a spelling error. He does not want the program to terminate, but wants a chance to reenter the name correctly.

The ERROR module of the General Purpose Modules allows recovery of errors caused by "non-catastrophic" conditions — cases that are not program logic errors but correctable conditions. (By catastrophe we don't mean Armageddon

or nuclear war — we mean a serious program error that ordinarily would cause the BASIC interpreter to stop execution.)

BASIC provides a command that will cause a user's error routine to be entered in case of **any** error. Executing an ON ERROR GOTO XXX command will cause a "jump" to the user's error routine at line XXX for any BASIC error. If the user's error routine is at line 11500, for example, executing ON ERROR GOTO 11500 will cause a transfer to line 11500 for every BASIC error.

What should be done in an error routine? The primary action in the routine should be to determine if the error is a program logic error or a correctable error. If the error is a logic error, then the routine will return control to the BASIC interpreter so that it can print out the error code and stop the program. If the error is correctable, then some type of warning message should be displayed and an appropriate correction action taken.

The sequence for returning to the BASIC interpreter in case of a logic error is to do the following:

- Execute an ON ERROR GOTO 0 statement. This "resets" the error "trapping" to the user's error routine.
- Execute a RESUME statement. This resumes execution at the line which caused the error. The error will occur again, but as the user's error routine is now not in control, the BASIC interpreter error "handler" will print out the error condition and stop program execution.

If the error is not a logic error then a warning message can be displayed, and a RESUME NEXT will execute the next line after the line in which the error occurred.

How does the user error routine know the type of error that has occurred? BASIC provides an error code which can be tested by using the ERR function. The error code can be compared with a list of acceptable error codes.

ERROR Module Operation

The ERROR module is shown in Figure 8-23. It uses the approach outlined above to test for recoverable errors. If a recoverable error is found, a warning message is displayed in the "prompt" message area, an error flag is set, and a return is made to the next BASIC line after the error. If a recoverable error is not found, a return is made to the BASIC error handler.

The ERROR module is entered after the error has occurred. At some point early in the application program an ON ERROR GOTO 11500 must have been executed.

Whenever a recoverable error is possible, the ZZ array must be initialized with all recoverable error codes. ZZ(0) contains the number of error code entries in the array. If no recoverable errors are possible, ZZ(0) is set to 0 and the ERROR module will **always** return to the BASIC error handler.

```
11500 GOTO 11640 'ERROR
11510 '*****************************************************
11520 ' THIS IS THE ERROR MODULE. IT RESPONDS TO A "NON-CAT-
11530 ' ASTROPHIC" SYSTEM ERROR BY OUTPUTTING A MESSAGE AND
11540 ' SETTING AN ERROR FLAG.
11550 '      INPUT: ZZ(0)=ZERO IF ALL ERRORS CATASTROPHIC,
11560 '                  NUMBER OF ERROR TYPES IF POSSIBLE
11570 '                  ERRORS
11580 '             ZZ ARRAY HOLDS ALL ERROR NUMBERS POSSIBLE
11590 '             ZZ$ ARRAY HOLDS ERROR MESSAGES
11600 '      OUTPUT:IF ERROR NUMBER FOUND, ERROR MESSAGE OUTPUT,
11610 '             XX FLAG SET, AND RETURN MADE TO NEXT STATE-
11620 '             MENT. IF NOT FOUND, RESUME AT ERROR.
11630 '*****************************************************
11640 IF ZZ(0)=0 GOTO 11710
11650    FOR XF=1 TO ZZ(0)
11660    IF XA=80 GOTO 11690
11670    IF ERR/2=ZZ(XF) GOTO 11740
11680    GOTO 11700
11690    IF ERR=ZZ(XF) GOTO 11740
11700    NEXT XF
11710 XB$="CATASTROPHIC SYSTEM ERROR":XB=3:GOSUB 11000
11720 ON ERROR GOTO 0
11730 RESUME
11740 XB$=ZZ$(XF):XB=3:GOSUB 11000
11750 XX=1
11760 RESUME NEXT
```

Figure 8-23. ERROR Module Listing

Each error code entry in ZZ has a corresponding error message in the ZZ$ array. The error code in ZZ(1) has an error message in ZZ$(1), the code in ZZ(2) has an error message in ZZ$(2), and so forth.

After a recoverable error code has been detected and the error message printed, ERROR sets variable XX to 1. This variable can then be tested after every action that might produce an error to see whether an error did occur.

An example of this is shown in the code of Figure 8-20 for CDLOAD. Error code 53 is stored in ZZ(1) and the message FILE NOT FOUND is stored in ZZ$(1). ZZ(0) is set to 1, for 1 entry. The OPEN file is then done. After the open, the XX flag is tested for a 1. If it is a 1, a FILE NOT FOUND error has occurred and the load is aborted by a RETURN without any INPUT action.

The first action in ERROR is to test for entries in the ZZ array. If there are no entries, line 11710 is executed. Line 11710 prints the message CATASTROPHIC SYSTEM ERROR and then resets the error "trap" by ON ERROR GOTO 0. The RESUME causes the line in which the error occurred to be executed once more causing the BASIC error handler to be entered (program stops).

If the ZZ array **has** entries, all entries are compared to the ERR code. If all entries are tested and no "match" is found, line 11710 is again executed to return to the BASIC error handling. If the ERR code is found in ZZ, the corresponding ZZ$ message is printed at line 11740. The XX flag is then set to 1, and a RESUME NEXT causes a return to the line following the line in which the error occurred.

Section III
An Application Example

Chapter Nine
MAILIST — Design Specification

In this section we'll design and code a typical business applications program using the General Purpose Modules as a base. MAILLIST will be an applications program that will process a file of mailing list entries — everything from initial entry, through disk storage, and on to label printing. While we don't claim that it is the ultimate mailing list program, we think you'll find that it works rather well with many features not found in other mailing list applications.

Using the Basic Plan

In Chapter 2 we described a basic plan in producing a business applications program. We'll try to follow the steps we discussed in that chapter here. Those steps were:

1. Learn the system characteristics
2. Learn the BASIC you'll be using
3. Research into the applications problem
4. Writing a Design Specification
5. General Program Design
6. Flowcharting the program
7. Coding the program
8. Entering the program
9. Debugging the program
10. Creating the final version of the program
11. Writing an operational manual for the program

We'll cover steps 1 through 4 in this chapter. In the other chapters of this section we'll cover the remaining steps for four separate sets of functions for the mailing list program.

Steps One and Two: Learning the System and BASIC

There's not a great deal that we can do for these two steps — they're really up to you. The MAILLIST application will be geared toward sequential disk file storage and processing using Level II and Disk BASIC. We can only recommend reviewing some of the Radio Shack manuals on specific items that might be giving you trouble.

As we did in the last section, we'll be explaining the actual BASIC code used in the program in detail. You may be able to jog your memory about certain BASIC statements and functions that are used by MAILLIST by the descriptive text. If operation of a BASIC command or statement is still fuzzy, go back and read the appropriate description in the Level II, Disk BASIC, or Model II BASIC manual.

As we mentioned earlier, these steps are "iterative" steps. Some people hate to wade through manuals without practical examples. They operate best by

learning enough to be dangerous, write some BASIC code, run into problems and **then** go back to find out what is happening. (The author admits to frequent use of this approach.) Other people are more disciplined — they go through an entire course in BASIC and then start writing programs. Use whatever approach works best for you.

Step Three: Research Into the Mailing List Problem

We've already done the research for you in this application. We wanted a mail list program that would operate as follows:

1. It would make efficient use of the General Purpose Modules, for obvious reasons.
2. All entries would be stored in RAM; although disk files would be used for storage, the entire disk file would have to be able to be contained in RAM. Disk file sorts and access were considered beyond the scope of this book.
3. Entries would not have fixed-length fields, to maximize the number of entries that could be stored.
4. Entries would be easily added, deleted, or modified.
5. Entries would be easily displayed with minimum search time.
6. Entries would be easily printed, in any format that the user desired.
7. Entries could be easily sorted on any field, although the main sort field would be last name.
8. A search could be made for any string of characters from a single letter or digit to a long name.
9. Up to eight separate fields could be used for each entry.
10. No assembly-language code would be used. Assembly-language code is excellent for certain portions of programs, but beyond the scope of this book.

Step Four: The Design Spec

The design specification for `MAILLIST` is shown in Figure 9-1. It includes all of the the operational procedures for `MAILLIST` in an example of "top-down" design. Of course, in this case, and in other applications that would use the General Purpose Modules as a base, the design is not strictly "top-down," as we were using the existing General Purpose Modules. We had to work within the framework of the array data storage, sequential disk files, menu format, display forms, and other characteristics of the General Purpose Modules.

In the general case, you might be defining the application from scratch, based on your research in step three and on your knowledge of the system and BASIC. You may choose to use the General Purpose Modules. If you do, you will be forced to accept the structure defined by the General Purpose Modules in return for ease of use and reduction in design time.

MAILLIST Design Specification

OVERVIEW:

MAILLIST is a complete mailing list program designed to operate on the TRS-80 Model I, II, or III Microcomputer Systems.

It can handle up to several hundred entries of forty or fifty characters each, or more entries if the entry size is reduced. MAILLIST builds an in-memory file of mailing list entries. Entries can be added, deleted, or modified at any time. High-speed "list" sorting is used to alphabetize each entry as it is added.

Entries may be displayed on the screen one at a time or in user-specified ranges. Entries may be printed on the system line printer in mailing label format or in any user-defined format, such as a line by line report.

Any character string, from a single character to larger strings, may be used in a "search" mode. MAILLIST will search all entries for the given string. This feature can be used to locate a street address, zip code, or other data.

Entries are arranged in alphanumeric order based upon the entire entry. JONES,ED,25555 OAKHURST appears before JONES,ED,25556 OAKHURST, for example. The "sort key" is essentially the last name. At any time the list of entries may be resorted based on any other field, such as zip code. This secondary sort can then be displayed or printed. This feature is handy for arranging the entries in zip code order for bulk mailing requirements, for example.

The complete list of entries may be saved as a disk or cassette (Model I/III) file with a user-specified name. Files can be loaded from disk, or two files may be merged together into a new "master file," again with user-specified name.

MAILLIST is "menu-driven" and uses a complete set of "fill-in forms" to prompt the user. Complete editing features are provided for entry of user data.

LOADING PROCEDURE:

To run MAILLIST, first load BASIC. If you are using the Model II, specify BASIC -F:1 to allow you to have at least one disk file. Next, load MAILLIST from disk by RUN MAILLIST, or load from cassette by CLOAD MAILLIST, followed by RUN. MAILLIST should start execution, and you should see the display shown in Figure 9-1-1.

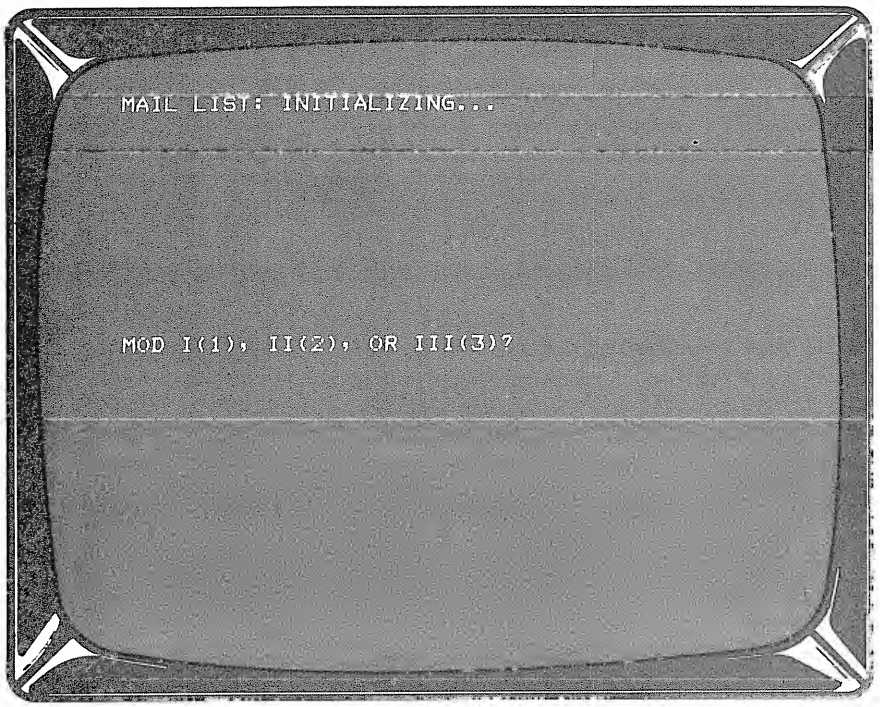


Figure 9-1-1. MAILLIST Initialization Display

Now enter the model number of your system, 1 for a Model I, 2 for a Model II, or 3 for a Model III. MAILLIST will continue initialization procedures as shown by an activity display in the upper right hand corner of the screen. There will be no display of the 1, 2, or 3 digit.

After initialization, MAILLIST will display a menu of items, as shown in Figure 9-1-2.

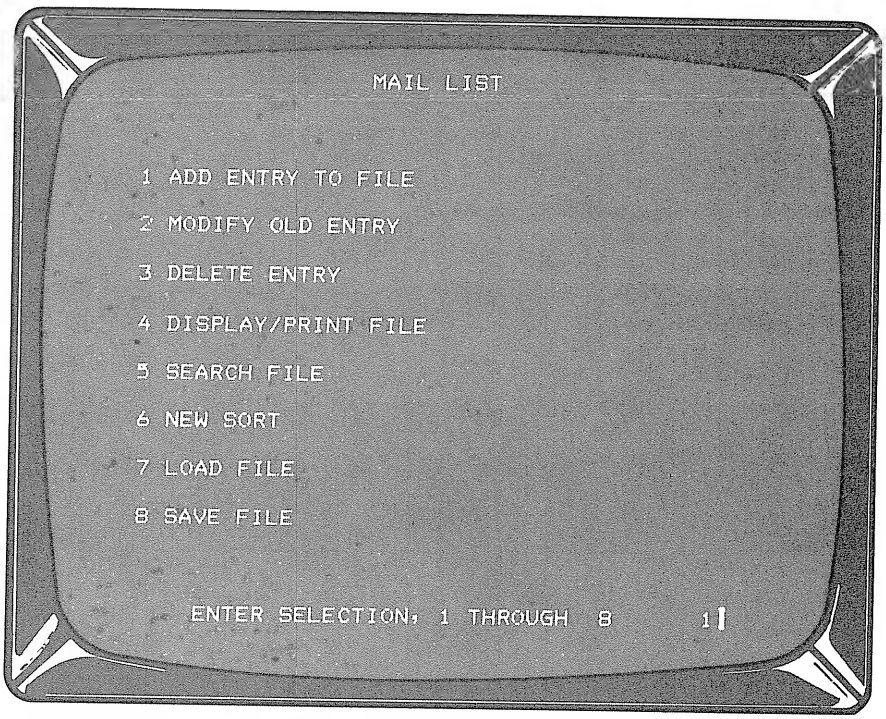


Figure 9-1-2. MAILLIST Menu

PROGRAM FUNCTIONS

The separate program functions for MAILLIST are shown in the menu of Figure 9-1-2.

Function 1 adds a MAILLIST entry to the file, or starts a new file of items.

Function 2 displays an existing entry by item number or name and allows a user to change part or all of the entry.

Function 3 displays an existing entry by item number or name and allows the user to delete the entry from the file.

Function 4 displays a range of items from the file. The range may be specified as a starting entry number or name and an ending entry number or name in any combination. This function also prints the range of items on the system line printer in mailing list or user-specified format.

Function 5 searches the file for a given search string. The first or all occurrences of the search string are displayed at the user's option.

Function 6 resorts the entire file based upon a user-specified "field."

Function 7 loads an old disk (or cassette) file. The file is loaded either as a new file, or is merged into the current file in memory.

Function 8 saves the current file on disk (or cassette).

DESCRIPTION OF FUNCTIONS:

Starting a New File or Adding an Entry

To start a new file or add an entry to the current file, select item 1 from the menu. You will see the display shown in Figure 9-1-3. This is the add entry form that must be filled in to add the entry. If a new file is being started, the entry will automatically start the new file.

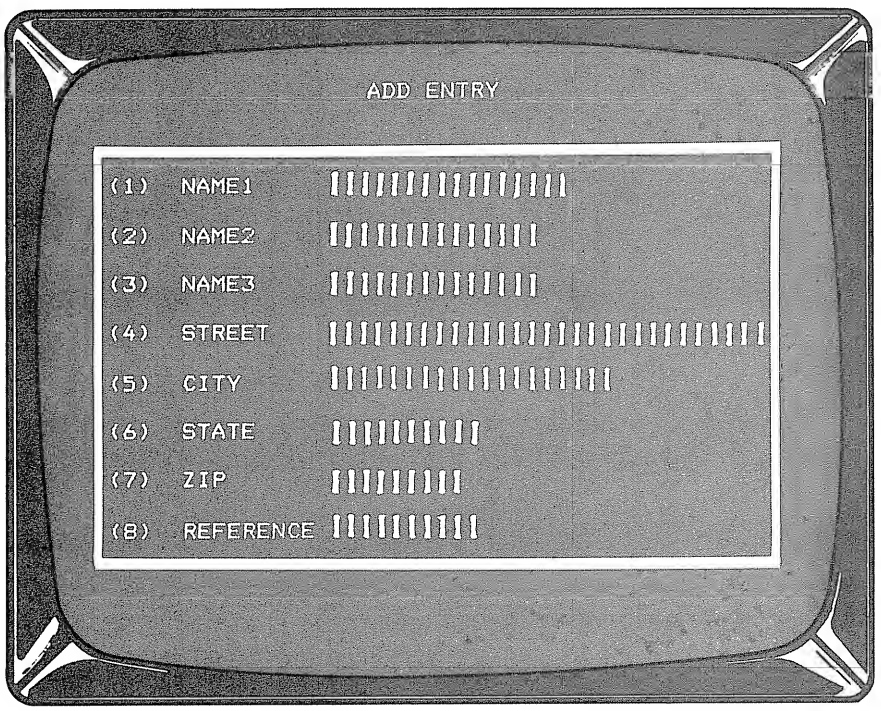


Figure 9-1-3. Add Entry Form

The form for add entry is the standard mailing list form for many other functions. It consists of eight "fields" — NAME1, NAME2, NAME3, STREET, CITY, STATE, ZIP, and REFERENCE. NAME1 is the primary "sort key." Use the last name of an individual for this field as all entries will be arranged in alphabetical order based upon this field. NAME2 is first name. NAME3 is company name, title, or other description or may be left blank. STREET, CITY, STATE, and ZIP are obvious. REFERENCE may be any particular coding you want to use for the entry — telephone number, date of entry, or other. As you may sort on this field by

the "secondary sort," you may wish to adopt a standardized coding of your own format.

Enter the data for each field, terminated by pressing the ENTER key. Any field may be left blank by simply pressing ENTER with no data. If at any time you wish to terminate the add operation, press CLEAR (Model I/III) or "up arrow" (Model II). This will take you back to the menu of items.

If you make a mistake, you may use the "left arrow" (Model I/III) or BACK-SPACE (Model II) to correct it. If you have made a mistake on the previous field, either terminate the add by CLEAR (Model I/III) or "up arrow" (Model II) or finish entering the data and perform a "modify" function.

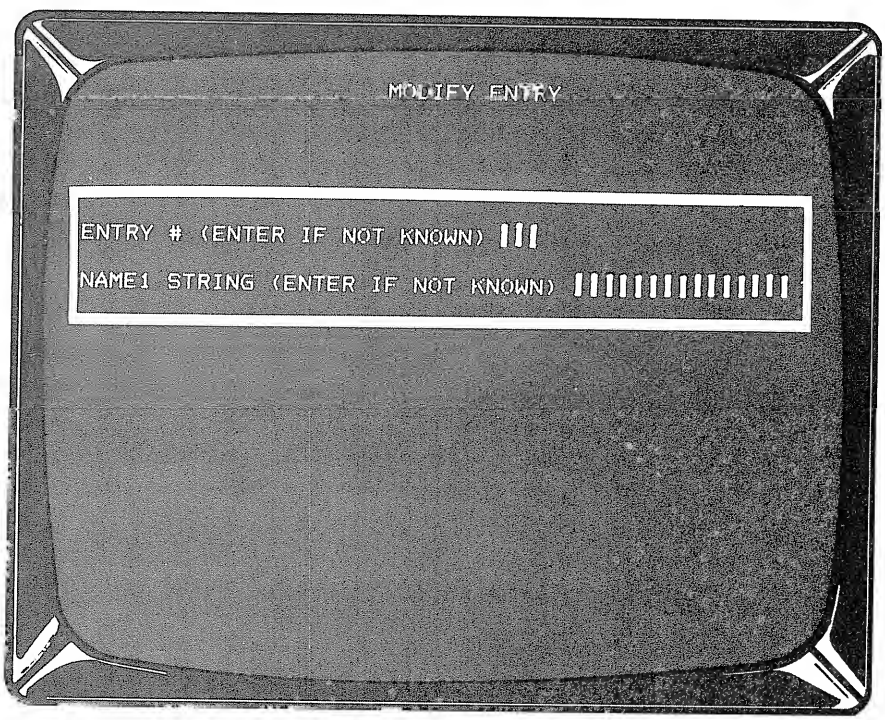


Figure 9-1-4. Modify Entry Form

Modifying an Entry

To modify an existing entry, select menu item 2. After 2 has been selected, you will see the form shown in Figure 9-1-4. Enter the entry number if known, say from a previous report. If the entry number is not known, press ENTER for the entry number and enter the last name of the entry in the next field.

MAILLIST will now search for the entry number or last name of the entry. After it finds the entry number or the last name, it will display the entry as shown in Figure 9-1-5, along with the message MODIFY YES OR NO. If you do not wish to modify the entry, enter an N. A return will be made to the menu.

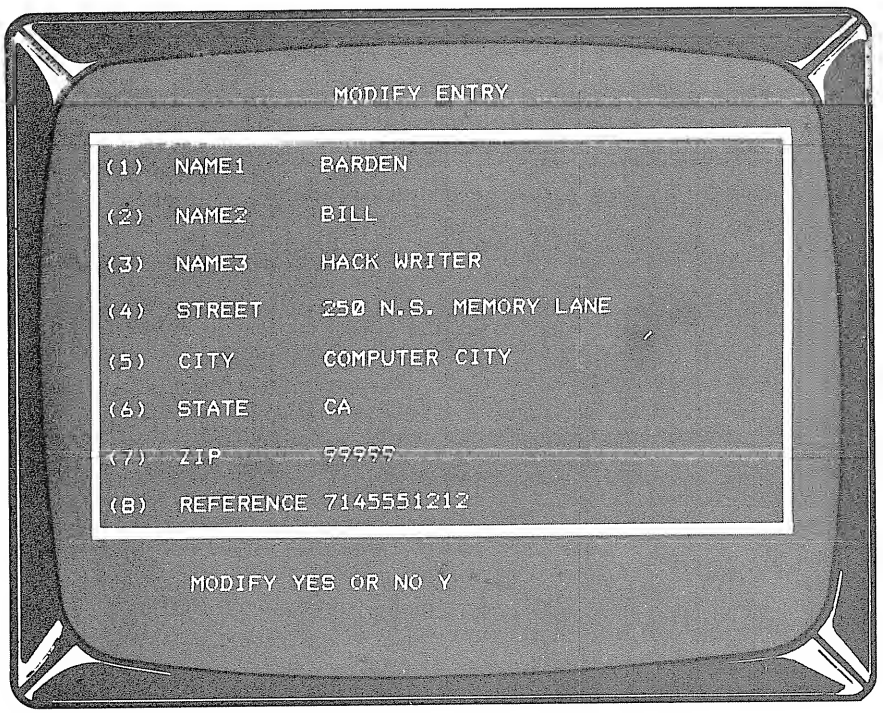


Figure 9-1-5. Modify Entry Display

If you do want to modify the entry, enter a Y. MAILLIST will now display the message FIELD # TO MODIFY. Scan the entry and enter the field number, 1 through 8, that you wish to modify. The selected field will be deleted, and you can reenter the data. Continue this process until the entry has been modified correctly. To terminate the modify, press ENTER after the field number "prompt." The entry will then be resequenced, based on the new information, and a return made to the menu.

At any time during the modify, CLEAR (Model I/III) or "up arrow" (Model II) may be pressed to return to the menu. If this is done after the MODIFY YES OR NO, however, the entry may be deleted.

Deleting an Entry

To delete an entry, select menu item number 3. The delete form will be displayed, as shown in Figure 9-1-6. Enter the entry number if known, or simply ENTER. Enter the last name of the entry if the number is not known.

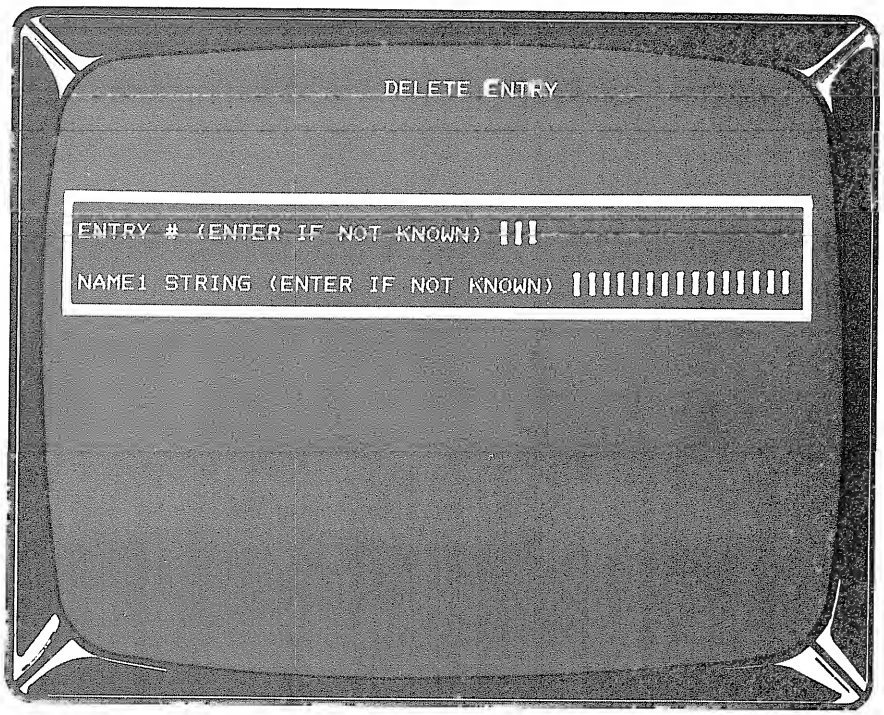


Figure 9-1-6. Delete Entry Form

MAILLIST will search for the entry number or last name and display the entry as shown in Figure 9-1-7, along with the message DELETE YES OR NO. If you wish to delete the entry, enter Y. The entry will be deleted and a return made to the menu. If you do not want the entry deleted, enter N for a return to the menu.

At any time, pressing CLEAR (Model I/III) or "up arrow" (Model II) will cause a return to the menu.

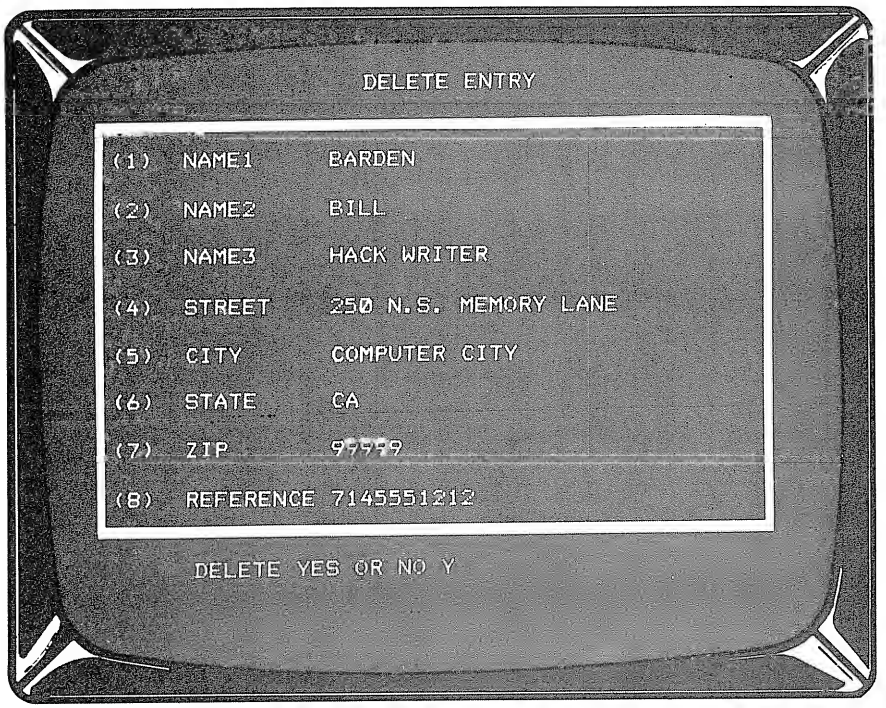


Figure 9-1-7. Delete Entry Display

Displaying an Entry

To display an entry or group of entries, select menu item number 8. The form shown in Figure 9-1-8 will be displayed. If the display is to be in order of NAME1, enter P. If the display is to be in the order of the secondary key, enter S. To display the entries in secondary key order, a secondary sort (function 6) must first have been performed; review this description.

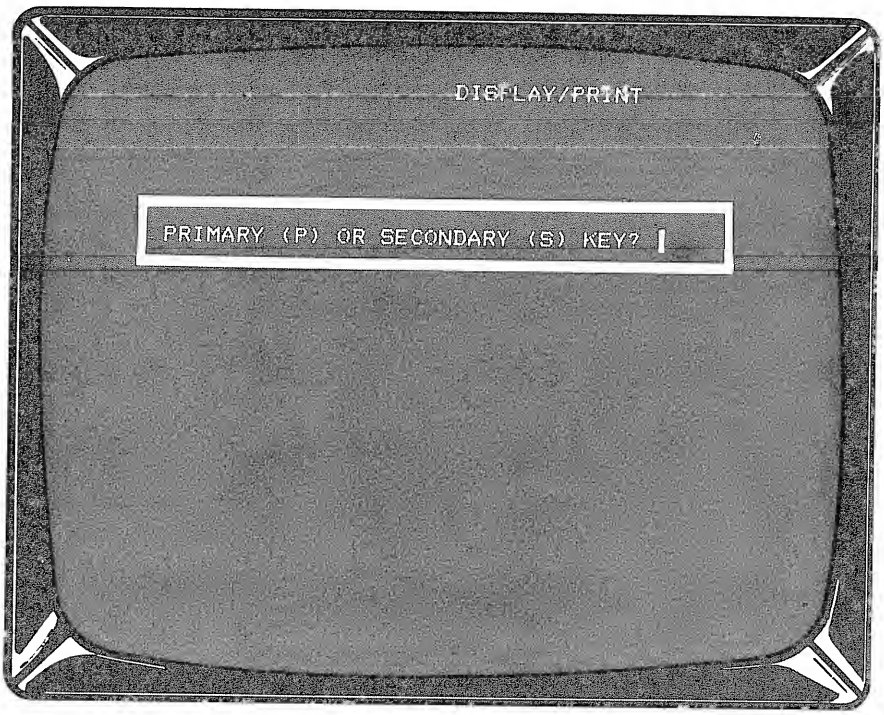


Figure 9-1-8. Display Entry Form

If the display is to be in primary key order, the form shown in Figure 9-1-9 will be displayed. If the display is to be in secondary key order, the form shown in Figure 9-1-10 will be displayed. Enter the start of the range by entering start number or name. Enter the end of the range by entering end number or name. If the start numbers or end numbers (or start names or end names) are not known, enter an approximate number or name. (The display may be interrupted at any time by pressing the CLEAR (Model I/III) or "up arrow" (Model II)). If the entire list is to be displayed, enter "1" for start number, followed by an ENTER for the other fields.

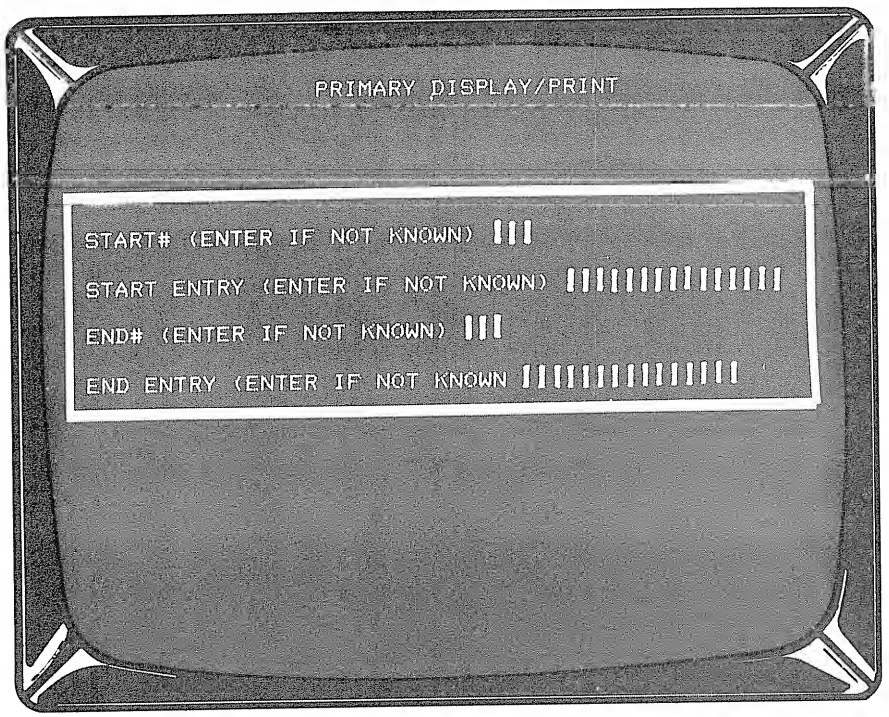


Figure 9-1-9. Primary Sort Form

MAILLIST will now search for the starting number or name and for the ending number or name. If found, the question DISPLAY(D) OR PRINT(P)? will be displayed. Enter D for display.

The range of entries in the list will now be displayed one at a time in standard format. An automatic return will be made back to the menu after the last entry has been displayed, or a return can be made by CLEAR (Model I/III) or "up arrow" (Model II).

The display function can be used to rapidly search the list for a desired entry by entering an approximate starting number or name and returning to the menu with CLEAR (or "up arrow").

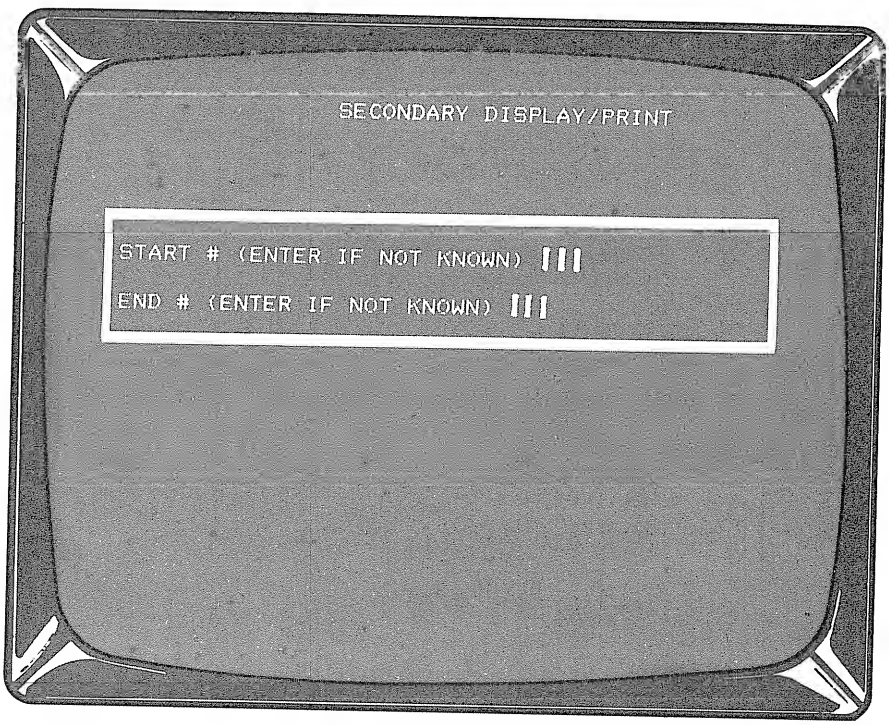


Figure 9-1-10. Secondary Sort Form

Printing an Entry

Menu item 4 also selects the print entry function. This works the same as the display function up until the question DISPLAY(D) OR PRINT(P)?. At this point, enter P.

MAILLIST will then ask CURRENT FORMAT (C) OR NEW (N)?. If you want a printout in standard MAILLIST format, enter C. Labels will be printed out as shown in Figure 9-1-11, for the entire range of entries selected.

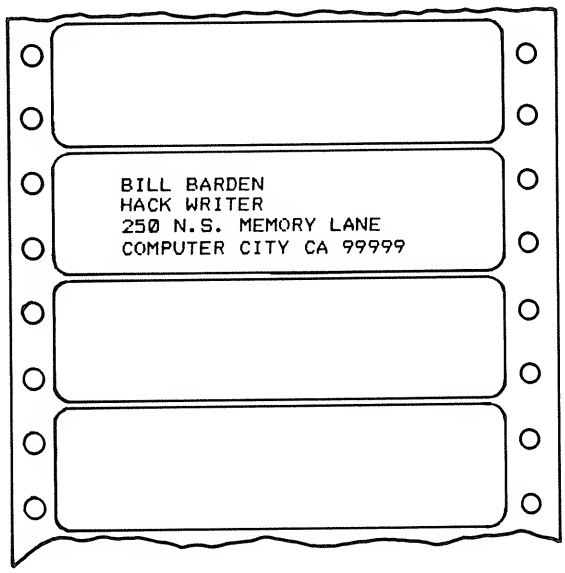


Figure 9-1-11. Label Printing

If you want a printout in a different format, enter N. MAILLIST will then ask SUPPRESS NEW PAGE, Y OR N?. If you want page formatting (no overprinting of the end of page) enter N, otherwise enter Y. Page formatting is not desirable for labels, for example, but is desirable for report formats.

MAILLIST will then display a list of entry items as shown in Figure 9-1-12, and then ask ITEM TYPE. These items define the format of the printout for each entry in the list. The set of items for the standard list is shown in Figure 9-1-13. To return to this format after definition of a new set of items, these items must be reentered.

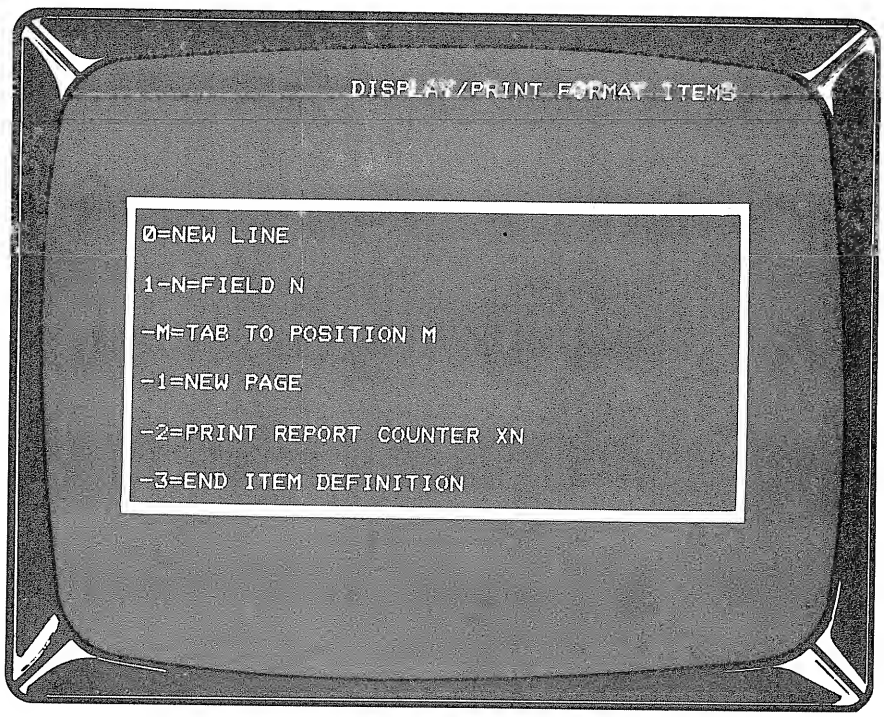


Figure 9-1-12. Standard Print Items

To define a new printout, enter a set of items after the "prompt" message. To end the entry, enter a -3. Some possible formats along with the standard format are shown in Figure 9-1-13.

**ONE INCH SPACING,
THREE LINE LABEL**

ENTER

```
BILL BARDEN
250 N.S. MEMORY LANE
COMPUTER CITY CA 99999
```

```
0,2,1,0,4,0,5,6,7,0,0,0,-3
```

SIMPLE REPORT

```
BARDEN BILL 250 N.S.
FOX KELLY 15321 OAKR
STOUT MICHAEL 921 E.
```

```
CA 99999
T CA 92153
92361
```

ENTER

```
0,1,2,3,4,5,6,7,-3
```

STANDARD LABEL

ENTER

```
BILL BARDEN
HACK WRITER
250 N.S. MEMORY LANE
COMPUTER CITY CA 99999
```

```
0,2,1,0,3,0,4,0,5,6,7,0,0,-3
```

Figure 9-1-13. Print Format Examples

After the new items have been defined, MAILLIST will print out the specified range of entries in the new print format. The CLEAR key (Model I/III) or "up arrow" key (Model II) may be used to end the printout at any time and return to the menu.

Search Processing

Selecting menu item 5 enters the search processing function. The form shown in Figure 9-1-14 is first displayed. Enter the search string; the string may be 1 to 30 characters.

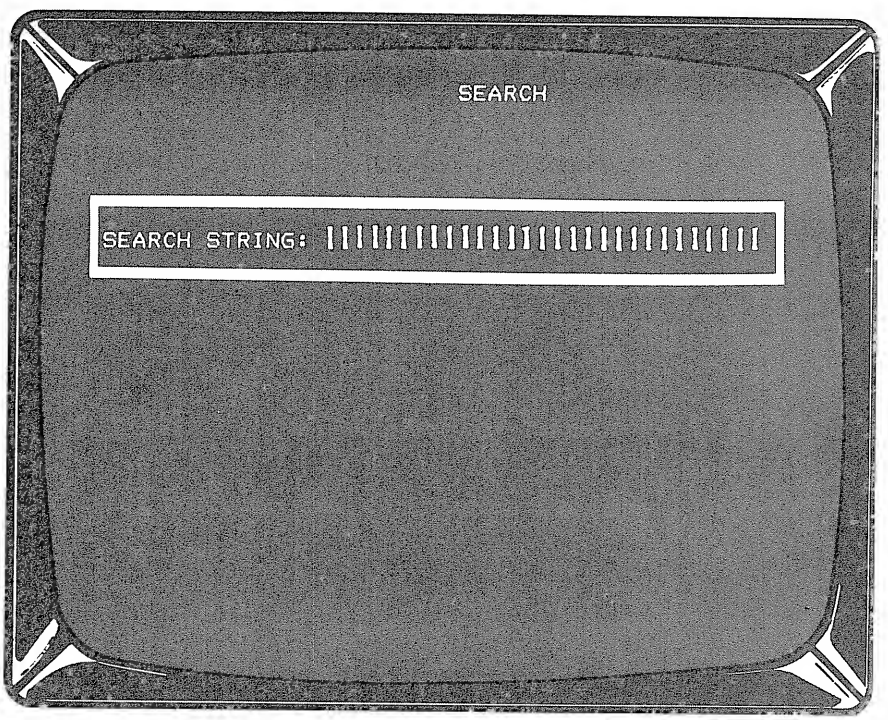


Figure 9-1-14. Search Processing Form

MAILLIST now searches the entire list, starting at entry 1. If the search string is found, the entry containing the search string is displayed as shown in Figure 9-1-15 and the program asks CONTINUE?. If you want MAILLIST to search for the next entry containing the string, enter Y, otherwise enter N to return to the menu.

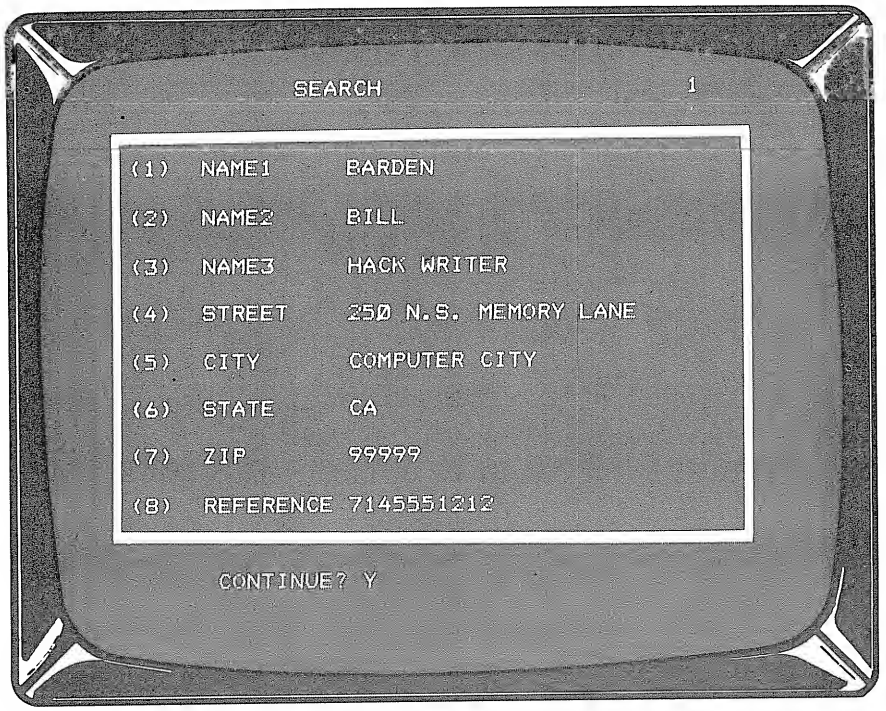


Figure 9-1-15. Search Display

When the entry containing the string is found (if it is found), the entry number is displayed in the upper right hand corner of the display.

Secondary Sort

To sort on another field than field 1, select menu item 6. After selecting the secondary sort, the form shown in Figure 9-1-16 will be displayed. Now enter the field number, 1 through 8 (field number 1 would only duplicate the currently sorted list, but it can be used).

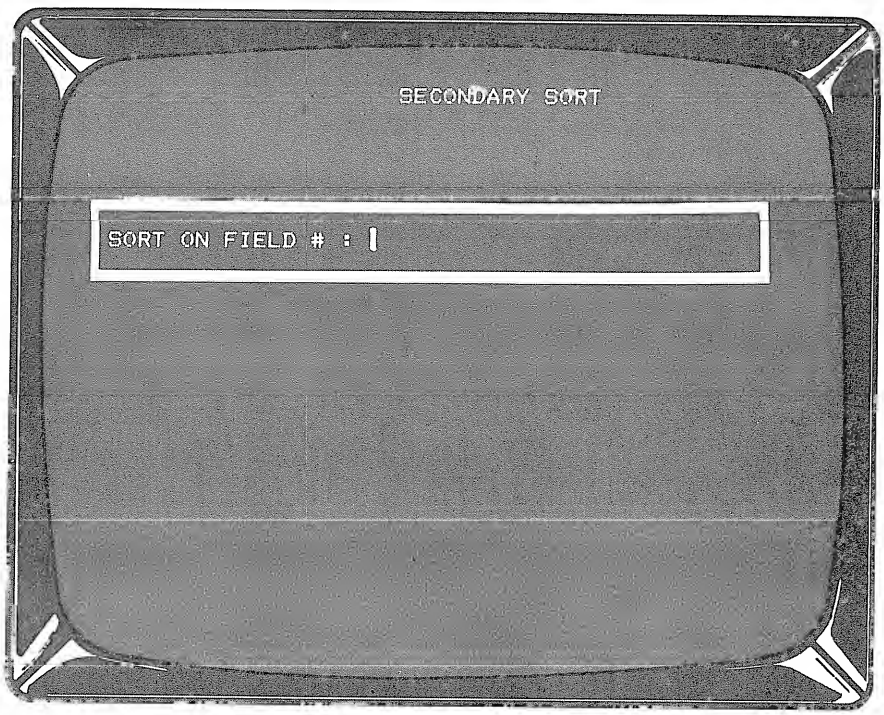


Figure 9-1-16. Sort Form

MAILLIST will now sort the entire file in memory, based on the selected field. This "sort" can be used in the display or print function to produce a list ordered on the selected field, as, for example, zip code order.

Because this sort is meant to give some versatility in ordering the entries without taking time during "add entry" processing, it is very slow compared to the interactive "add entry" sort. Expect the secondary sort to take one hour for 100 entries and longer times for larger files. The "activity" of the sort is indicated by a display in the upper right-hand corner of the screen.

Load File

Menu item 7 selects the load file function to load a MAILLIST file previously written out by the Save File function (see next item description). The form displayed in Figure 9-1-17 is displayed for this processing. If the file to be loaded is on disk, enter D, otherwise enter C for cassette. The disk filename must be entered for disk files but can be ignored for cassette files; simply type in ENTER in the latter case.

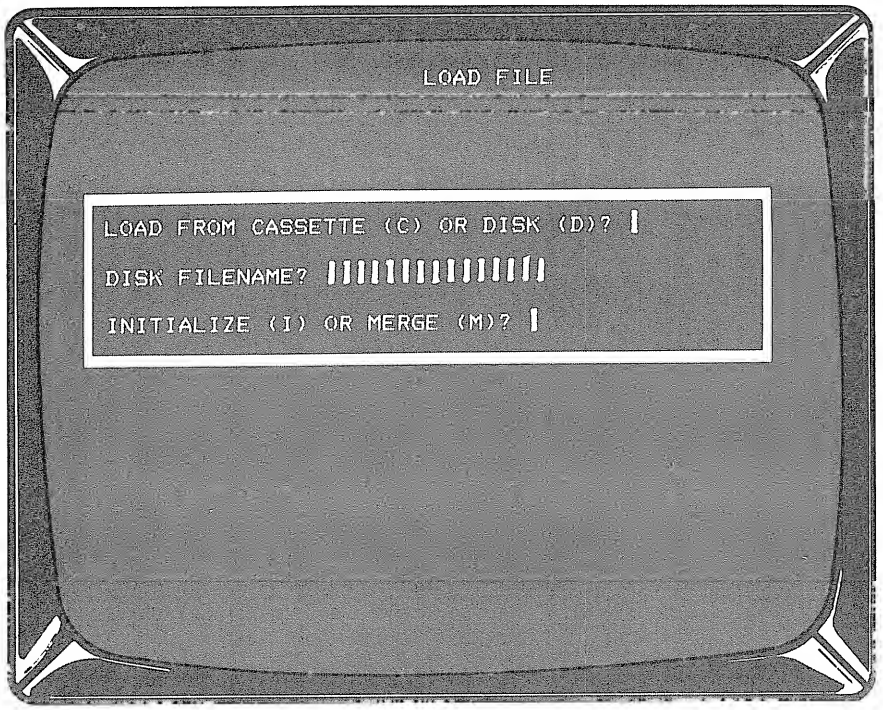


Figure 9-1-17. Load File Form

If you want the file to be loaded in by itself and do not want to retain the current entries in memory, enter "I" for initialize. In this case the file will load in and "overlay" any existing entries.

If you want the file to be "merged" with current entries in memory, type "M" for merge. In this case the file will be loaded in and merged with the entries already in memory. The result will be a new file in memory with all entries in correct order. The merge function can be used to consolidate two files on disk or one on disk (cassette) and one in memory.

If a cassette file is being initialized or merged, position the cassette tape just prior to the file before entering the I or M. MAILLIST will read the first file that it finds on cassette.

Save File

Menu selection 8 selects the save file function. The form shown in Figure 9-1-18 is displayed for save file. Enter C for cassette or D for disk. Enter an appropriate file name for a disk file; this field will be ignored for cassette files (type ENTER).

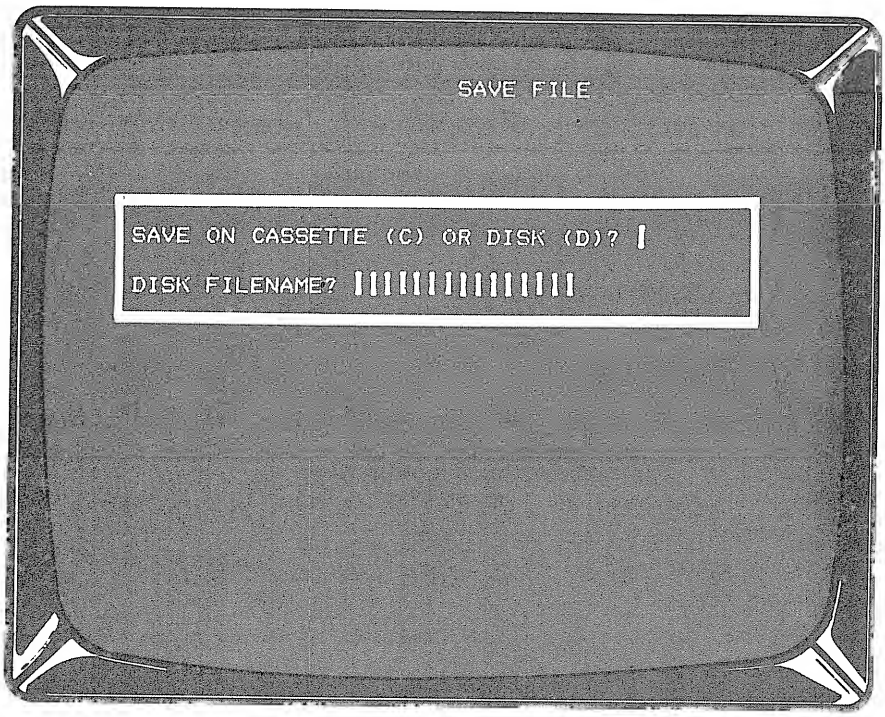


Figure 9-1-18. Save File Form

MAILLIST will then save the entire file in memory on disk or cassette, under the specified file name. A return is then made to the menu.

SYSTEM ERROR MESSAGES:

During the course of MAILLIST processing, certain error messages may be displayed. Here is a list:

Add Entry

OUT OF MEMORY: Current entries exceed memory limits.

Modify/Delete Entry

REENTER VALID NUMBER OR NAME: Erroneous entry number or name used in specifying entry for modify.

Display/Print Processing

NEVER SORTED OR MODIFIED - RESORT: Secondary sort has never been performed, or entries have been added, deleted, or modified after a secondary sort.

START NUMBER NOT FOUND: Starting number exceeds number of entries in list.

NO START NUMBER OR STRING: Neither a number nor name was entered.

START STRING NOT FOUND: Name was beyond last entry in list.

END STRING NOT FOUND: Name was beyond last entry in list.

INVALID ITEM TYPE: Invalid print format item was entered.

Load File

FILE NOT FOUND: Disk file was not found on diskette.

MAILLIST DISK FILES:

MAILLIST disk files are ASCII files. They may be LISTed and PRINTed as any ASCII file as a means to examine mail list files without using the MAILLIST program.

Chapter Ten
MAILLIST—MAIN Driver

In the previous chapter we produced a design specification that defined what MAILLIST was to do, but we did not include any information on **how** to do it. In this chapter and following chapters we'll define how the MAILLIST functions will be implemented by a series of **flowcharts** and the actual code for MAILLIST.

In most programs, the processes of steps 5 and 6 — general program design and flowcharting — are **iterative**. When we start thinking about how things are to be done and flowcharting the process, we see errors in the way program functions are to proceed. We also see more efficient ways of doing things! Programmers are naturally lazy, and if it's possible to change the design specification slightly to make the program easier to write, it's no sin. (It's best not to change the "scope" of the design spec, however, say from a mail list function to accounts receivable!)

Step 5: General Program Design

The general program design in MAILLIST has been greatly simplified by using the General Purpose Modules as a base. We already have the program routines to add and delete entries to a file of data, to save and load the file of data to disk or cassette, to generate forms and input data to the forms, to search the file, and other functions. The **structure** of the file is also well defined; this is good **and** bad, of course. We are "locked-into" a specific data structure that uses a list of items in an array.

We have already done a major part of the program design in writing the design specification. We drew on our past experience either consciously or subconsciously about what could feasibly be done in the program. We knew, for example, that it was no problem to produce a form on the screen, or to input character data to the form, and included such functions in the design specification.

Because we're using the General Purpose Modules, we can proceed right to the next phase of the implementation, the flowcharting of MAILLIST. We'll cover any unanswered questions about general design in this step.

Step 6: Flowcharting MAILLIST

We'll be using a few standard flowcharting symbols in the flowcharts of this section. If you've forgotten what they represented, go back to Chapter 2 and review the descriptions. There's nothing mysterious about using the symbols — they're just a shorthand way of showing the program flow so that we can then convert to BASIC statements.

First of all, assume that we have the use of all of the General Purpose Modules. We'll be using most of them, all of them in fact, as **subroutines** to simplify our own coding. In the flowcharts we'll refer to them by the  symbol and use their titles **and** line numbers.

The flowcharts will generally be divided into separate functions that are related to each of the eight functions of the MAILLIST program as defined in the design specification. You'll find that most business applications programs can be broken down this way, and it's very convenient to be able to **segment** a large program in this fashion.

Step 7: Coding MAILLIST

At the same time that we show the flowcharts for MAILLIST, we'll show the actual code. In actual practice, the flowcharts might have been done before the code, at the same time as the code, or not at all, as we described in Chapter 2.

The General Purpose Modules have lines numbered below 20000, as shown in Figure 3-5. We will use lines 20000 and above for the MAILLIST "modules." We say modules here, because the MAILLIST application can be as easily divided into segments just as the General Purpose Modules were.

The line numbering scheme we'll use for the MAILLIST modules will be similar to the GPM. We'll start each module at some increment of 500, such as 23000 or 23500, starting at line 20000. We'll use line "increments" of 10, so that the line numbering runs 23000, 23010, 23020, and so forth. This is just a nicety, and you can use whatever increment you wish in your applications program. (If you feel more at ease in increments of 16 (hexadecimal), you have great promise as an assembly-language programmer!)

We'll also follow a "standard" format for the MAILLIST modules. On the first line we'll have a GOTO followed by the title of the module. An example is 20000 GOTO 20010 'MAIN. The GOTO bypasses the descriptive text at the beginning of the module, which will be deleted in the "compressed" form of MAILLIST. Here again, the MAILLIST modules will be shown as "uncompressed" with REMarks and blanks for appearance. We'll also provide a compressed format for high-speed and minimum memory in Appendix II.

The Main Driver

The flowchart of MAIN, the main **driver** of MAILLIST is shown in Figure 10-1 and the corresponding code is shown in Figure 10-2. MAIN's purpose is to start the MAILLIST processing, to display the menu of items (functions) available, and to **branch out** to a selected function. After the function has been completed, MAIN will be **reentered** to repeat the process for the next selected function. MAIN is really the main **loop** of the program, and for that reason we call it a "driver." This approach is a tried-and-true approach to use in any large program, and we can't go too far wrong in using it for MAILLIST.

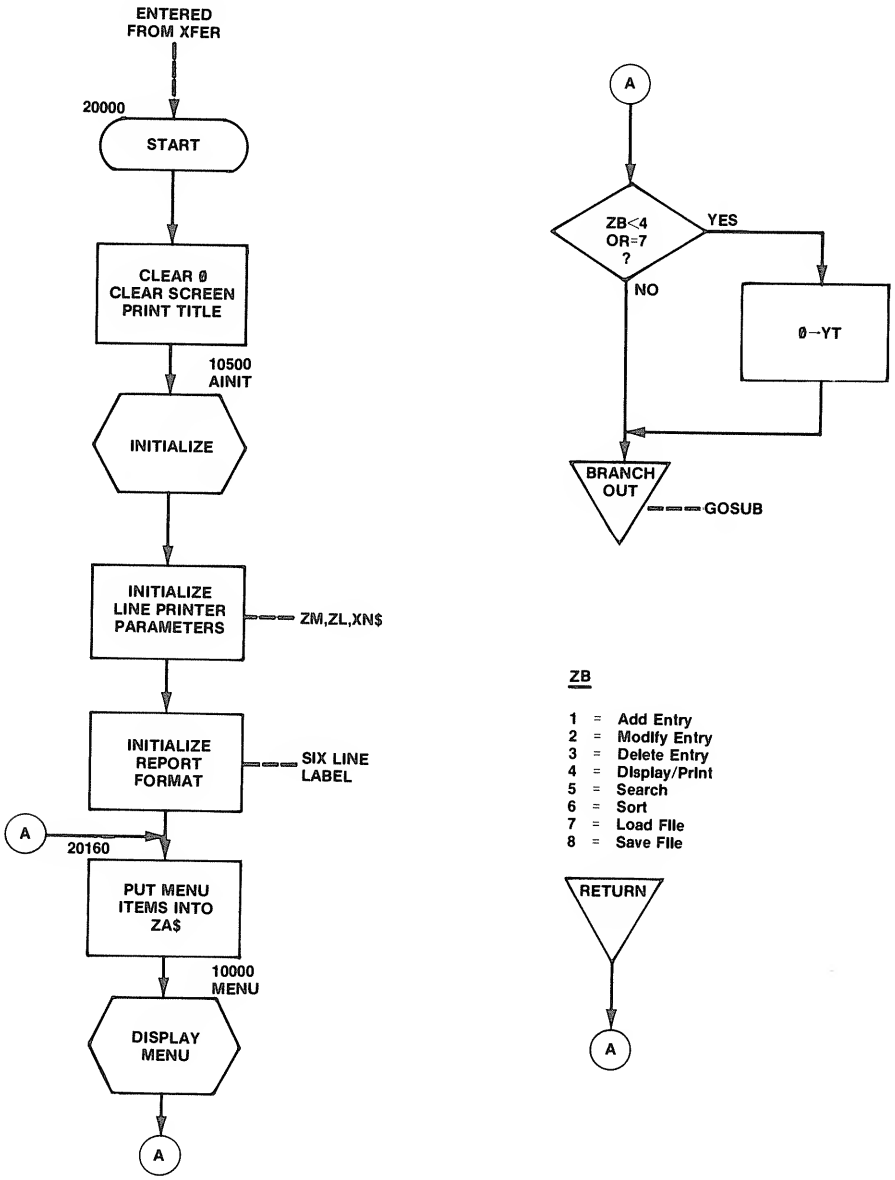


Figure 10-1. MAIN Module Flowchart

The very first thing we want to do in MAIN is to call the AINIT module at line 10500. AINIT must be called **once** at the beginning of each applications program, but never again! We could just call AINIT without doing anything else but to comply with the design spec, we're going to output a message about

```
20000 GOTO 20010 'MAIN
20010 CLEAR 0
20020 '*******************************************************
20030 ' THIS IS THE MAIN DRIVER. IT CONTROLS ALL PROCESSING
20040 '*******************************************************
20050 'OUTPUT TITLE
20060 CLS:PRINT @ 0,"MAIL LIST: INITIALIZING...";
20070 'CLEAR STRING SPACE AND INITIALIZE ARRAYS
20080 GOTO 10500
20090 'INITIALIZE LINE PRINTER PARAMETERS
20100 ZM=-1:ZL=66:XN$="                                    MAIL LIST"
20110 'INITIALIZE REPORT FORMAT
20120 XP(0)=13:XP(1)=0:XP(2)=2:XP(3)=1:XP(4)=0:XP(5)=3
20130 XP(6)=0:XP(7)=4:XP(8)=0:XP(9)=5:XP(10)=6:XP(11)=7
20140 XP(12)=0:XP(13)=0
20150 'DISPLAY MENU
20160 ZA=8:ZA$(0)="MAIL LIST":ZA$(1)="ADD ENTRY TO FILE"
20170 ZA$(2)="MODIFY OLD ENTRY":ZA$(3)="DELETE ENTRY"
20180 ZA$(4)="DISPLAY/PRINT FILE":ZA$(5)="SEARCH FILE"
20190 ZA$(6)="NEW SORT":ZA$(7)="LOAD FILE":ZA$(8)="SAVE FILE"
20200 GOSUB 10000
20210 'ZB CONTAINS SELECTION VALUE OF 1-9. NOW BRANCH OUT.
20220 'RESET SECONDARY SORT FLAG IF PENDING CHANGE
20230 IF ZB<4 OR ZB=7 THEN YT=0
20240 ON ZB GOSUB 20500,21000,21500,22000,23000,23500,24000,24500
20250 GOTO20160
```

Figure 10-2. MAIN Module Listing

initialization. The reason for this is that the initialization process in AINIT does take some time, and it's reassuring to the user to see something happening. Don't forget also that AINIT shows "activity" in the "activity area" of the screen to indicate that some processing is taking place.

AINIT is called by a GOTO 10500. Remember that this is a special case of a module that is not called by a GOSUB as AINIT performs a CLEAR that erases the return point. The return point here is at line 20100. A GOTO 20100 is the last action of AINIT, and line 20100 must be used in every applications program for the return, unless the GOTO in AINIT is changed.

Before calling AINIT we perform a CLEAR 0. This is an optional step to release all previously allocated string space to AINIT. If MAILLIST is **restarted**, this step brings us back to a known point.

The next step is to initialize all system variables and parameters that were not initialized in AINIT. This would include:

- REPORT parameters that control the REPORT printing
- Line printer parameters for page length, number of lines per page, and automatic page titling
- Error conditions in ERROR for allowable errors with proper messages (not done here)

AINIT handles everything else for us. Depending upon the applications program, you may have other variables that must be initialized at this point.

Remember that we're going to perform these functions only one time, and that these are **initialization** procedures only.

Now we come to the section of MAIN that is the **reentry point** (line 20160). This point will be reentered after the menu function has been completed. The first thing we want to do here is to clear the screen and output the menu of items by using MENU. Before we can do this we must define the parameters that MENU uses, the number of items in ZA, and the text for the items in string array ZA$. After these are defined, MENU is called at line 10000. Calling MENU clears the screen and automatically displays the menu items, properly centered with menu title. It also displays the title message in ZA$(0).

On return from MENU, ZP holds the selection value of 1-8. A return isn't made unless the user has input the proper value. The selection value corresponds to the number of the function in the ZA$ array. The next action is to "branch out" to the processing for the function. We could say `IF ZP=1 THEN GOSUB 20500` and so on, but it's much more convenient to use ZB in a "computed GOSUB." This statement will perform a GOSUB 20500 for ZB=1, a GOSUB 21000 for ZB=2, and so forth. After the function processing has been completed, it will RETURN to the next line, which will take us back to the reentry point at 20160.

The line numbers for the GOSUBs were determined **after** writing the code for each function. Note that some of the line numbers for the functions jump by 1000 instead of 500. The reason for this is that these turned out to be larger modules and we could not maintain both line increments of 10 and contain the code in 50 lines. We could have gone to smaller increments, but this is a problem when using one large program and using the **renumbering** capability provided in the Renumber utility program.

Before the "branch out" to the MAILLIST function was made, we first performed a test of ZB and set variable YT on the result. YT is the "secondary sort" flag used in the SECSRT module. If it is 0, it means that either no secondary sort was done, or that an add entry, delete entry, modify entry, or load file function invalidated the previous sort. In this case the sort has to be done again before a display or print of the list on the secondary "key" can be done. Here we test for an add, modify, or delete by testing for a MENU item selection of less than 4 or for a load file function of 7. If any of these are done, then YT is reset to 0. If none of these functions are to be performed, then the secondary sort will still be valid after the function (display/print, search, new sort, or save file). YT is 0 right after MAILLIST load, so that condition is also logically correct.

In the following chapters we'll discuss the other MAILLIST functions, roughly grouped by functions of similar nature.

Chapter Eleven
MAILLIST—Adding, Deleting and Modifying Entries

In this chapter we'll discuss the flowchart and code of three MAILLIST functions — adding an entry to the file, modifying an entry in the file, and deleting an entry from the file. A branch is made to one of these functions from the MAIN driver discussed in the previous chapter. All three modules of MAILLIST draw heavily on the General Purpose Modules (as one might suspect at this point).

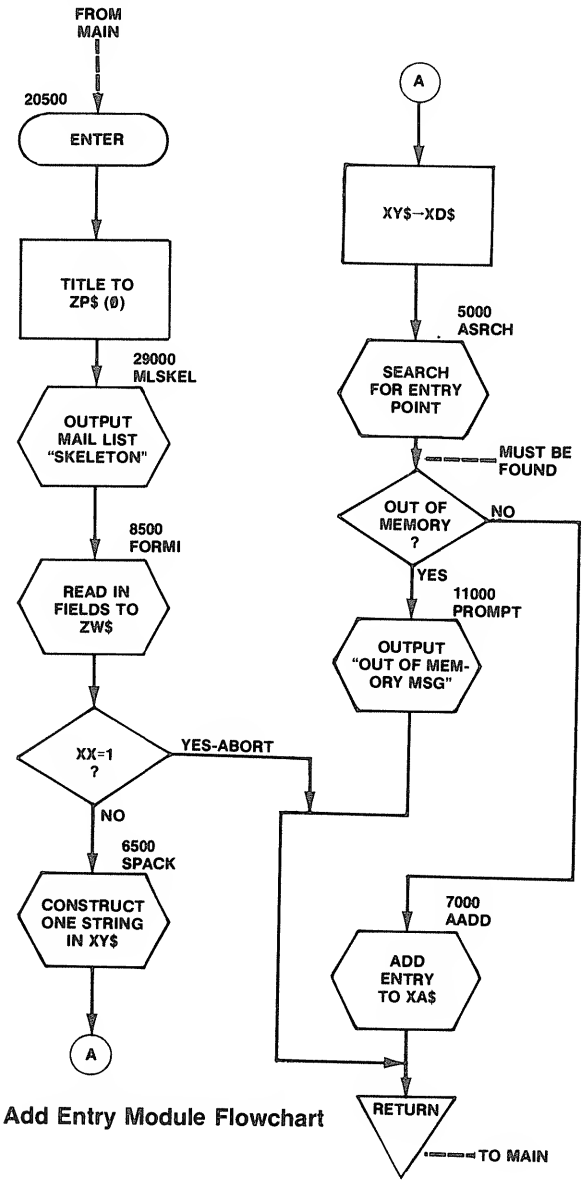


Figure 11-1. Add Entry Module Flowchart

Add Entry Processing — MFADD

The flowchart for Add Entry is shown in Figure 11-1, while the actual code for this function is shown in Figure 11-2.

```
20500 GOTO 20550 'MFADD
20510 '*************************************************************
20520 ' THIS IS ADD ENTRY PROCESSING.
20530 '*************************************************************
20540 'FIRST OUTPUT SKELETON
20550 ZP$(0)="ADD ENTRY":GOSUB 29000
20560 'NOW READ IN FIELDS
20570 GOSUB 8500:IF XX=1 GOTO 20680
20580 'NOW CONSTRUCT ONE STRING FROM FIELDS
20590 GOSUB 6500
20600 'NOW SEARCH FOR KEY IN EXISTING FILE
20610 XD$=XY$:GOSUB 5000
20620 'IF NOT OUT OF MEMORY, CONTINUE
20630 IF XM<>2 GOTO 20670
20640 XB=3:XB$="OUT OF MEMORY":GOSUB 11000
20650 GOTO 20680
20660 'NOW ADD ENTRY
20670 GOSUB 7000
20680 RETURN
```

Figure 11-2. Add Entry Module Listing

These are the steps in adding an entry to the existing file in RAM:

1. Output an add entry form
2. Input the fields for the form
3. Construct one large string from the individual fields
4. Add the string in the proper place in the array in memory
5. Go back to the MAIN program for the next function

All of the functions above can be done by calling a GPM module. We have a module to output a form (FORMO), a module to read in the fields of the form (FORMI), a module to "pack" the fields (SPACK), a module to search for the insertion point (ASRCH), and a module to add the entry (AADD). It should be an easy job to tie these together to implement the add function (all right, then, an easy job compared to, say, a 4000-person payroll program . . .).

First of all we must clear the screen and output the add entry form. The form is shown in Figure 11-3. Although we could call FORMO after setting up the field descriptions and lengths in the ZP$ and ZR arrays, we'll take another tack. Since this form is continually being output, we'll make our own subroutine (module). This is shown in Figure 11-4; it's called MLSKEL, "Mail List Skeleton," and calls FORMO to clear the screen and output the skeleton. We've put it out of the way starting at line 29000.

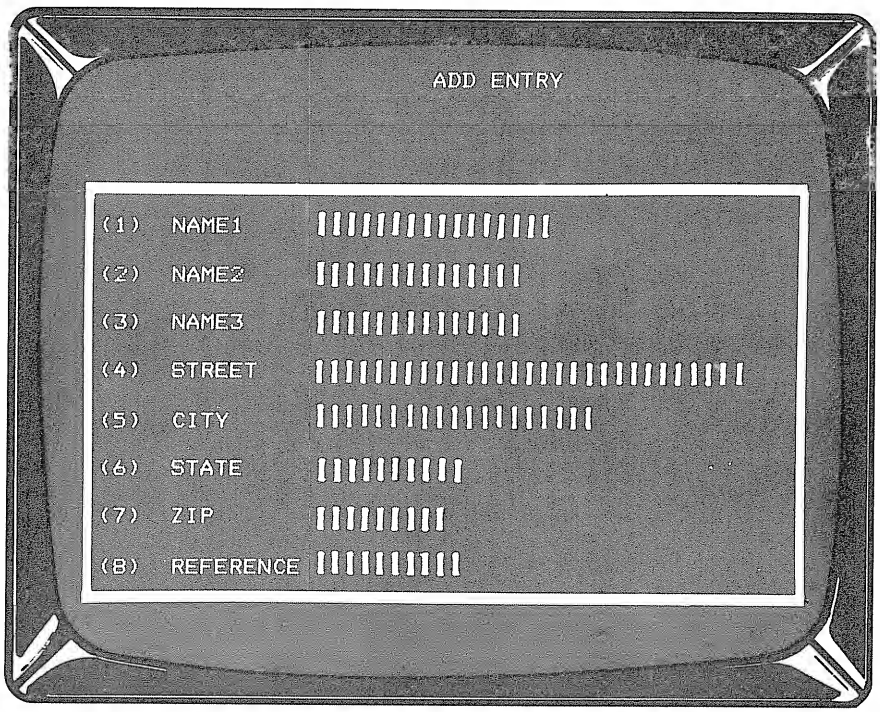


Figure 11-3. Add Entry Form

```
29000 GOTO 29060 'MLSKEL
29010 '*****************************************************
29020 ' THIS MODULE DISPLAYS THE MAIL LIST SKELETON. IT IS USED
29030 ' EVERY TIME THE STANDARD FORM IS TO BE DISPLAYED.
29040 ' ZP$(0) MUST CONTAIN THE TITLE
29050 '*****************************************************
29060 ZP=57:Z0=8
29070 ZP$(1)="(1)  NAME1     ":ZP$(2)="(2)  NAME2     "
29080 ZP$(3)="(3)  NAME3     ":ZP$(4)="(4)  STREET    "
29090 ZP$(5)="(5)  CITY      ":ZP$(6)="(6)  STATE     "
29100 ZP$(7)="(7)  ZIP       ":ZP$(8)="(8)  REFERENCE"
29110 ZR(1)=17:ZR(2)=15:ZR(3)=15:ZR(4)=30:ZR(5)=20
29120 ZR(6)=10:ZR(7)=9:ZR(8)=10:GOSUB 8000
29130 RETURN
```

Figure 11-4. MLSKEL Module Listing

MLSKEL displays all parts of the standard form except for the title in ZP$(0). Since this will be different for different functions, we'll leave that as one of the parameters to be set before MLSKEL is called. The first thing we'll do, then, is to set ZP$(0) to ADD ENTRY as a title and call MLSKEL at 29000.

We now have the form on the screen. Next, the user must fill in the form by entering data for each field. The FORMI does this automatically, and returns field data in ZW$(1) through ZW$(8). FORMI is called by a GOSUB 8500. After the RETURN is made, variable XX may be set to a 1 if the user wanted to terminate the add operation. In this case we must return to the menu without taking any action, so we'll test XX and return if XX=1. No harm is done in this action as the list in memory hasn't been altered in any way.

Now we've got the data on the screen and in the ZW$ array. To add the entry to the list, we've got to "pack" it into one string. To do that we'll call SPACK at line 6500. SPACK takes the data from ZW$ and puts it into standard GPM format in string XY$. A sample of this operation is shown in Figure 11-5. After return is made from SPACK, XY$ contains the packed string.

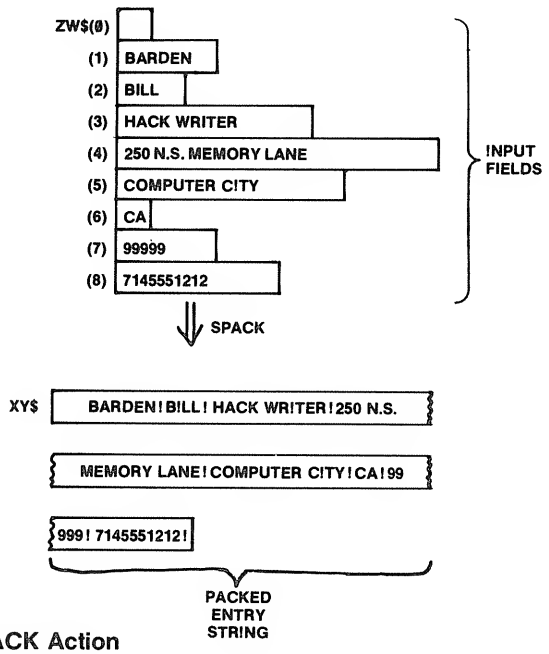


Figure 11-5. SPACK Action

Now that we have the entire entry, we must search for the proper entry point in XA$, using the pointers in XA%. The ASRCH module does this for us, but works with a string in XD$. We must first transfer XY$ to XD$. After we do that, we can call ASRCH at line 5000.

ASRCH returns XJ, XK, and XL along with some other parameters to indicate how the search went. In the case of an add, we don't expect to find the entry, and XJ, XK, and XL will be set to the proper insertion point even if the list is empty. If the entry is found, this means we've already added the entry previously. We make no provision for notifying the user about this, although we could have. It will be apparent to the user soon enough if this is the case, and a delete is easy.

About the only thing we have to watch for is being out of memory because of too many entries. In this case we won't (can't) add the entry. We **will** output a warning message by calling the PROMPT module, however. If XM=2, the message OUT OF MEMORY will be printed at the prompt message area, and we'll simply return to the menu.

If XM is not 2, we can go ahead and add the entry. At this point XJ, XK, and XL are set to their proper values for the add. The AADD module is set up to use these parameters after an ASRCH call, so we can simply call AADD by a GOSUB 7000. The entry string in XD$ will be added to the list in memory at the proper place.

After a RETURN is made from AADD, a RETURN is made from the MFADD module, and the MAIN driver is reentered.

It's important to note which variables are being used in this process and to realize that they are **non-conflicting**. Certain variables "ripple through" several modules, being used for a succession of actions. A typical add with variable action is shown in Figure 11-6.

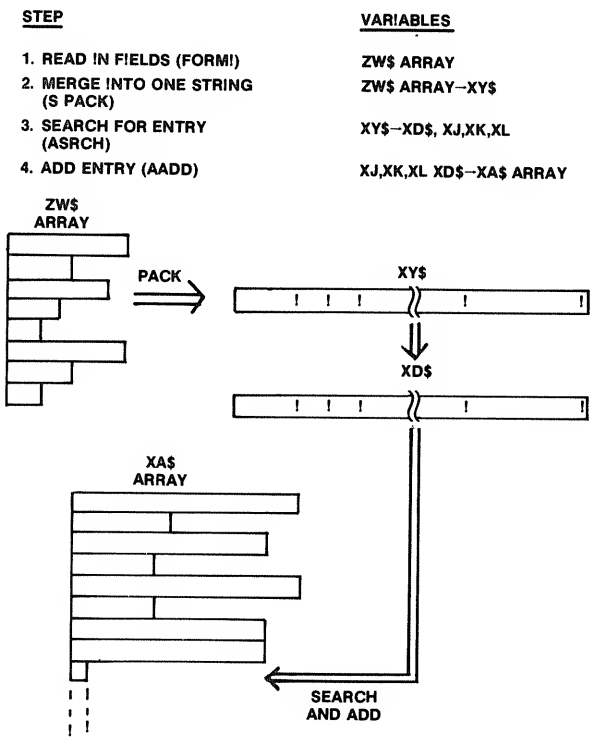


Figure 11-6. Typical Add Entry

Delete Entry Processing — MFDEL

Figure 11-7 shows the Delete Entry flowchart, and Figure 11-8 shows the actual code. We'll discuss this module first, as the Modify Entry uses a portion of MFDEL to avoid repetitive code.

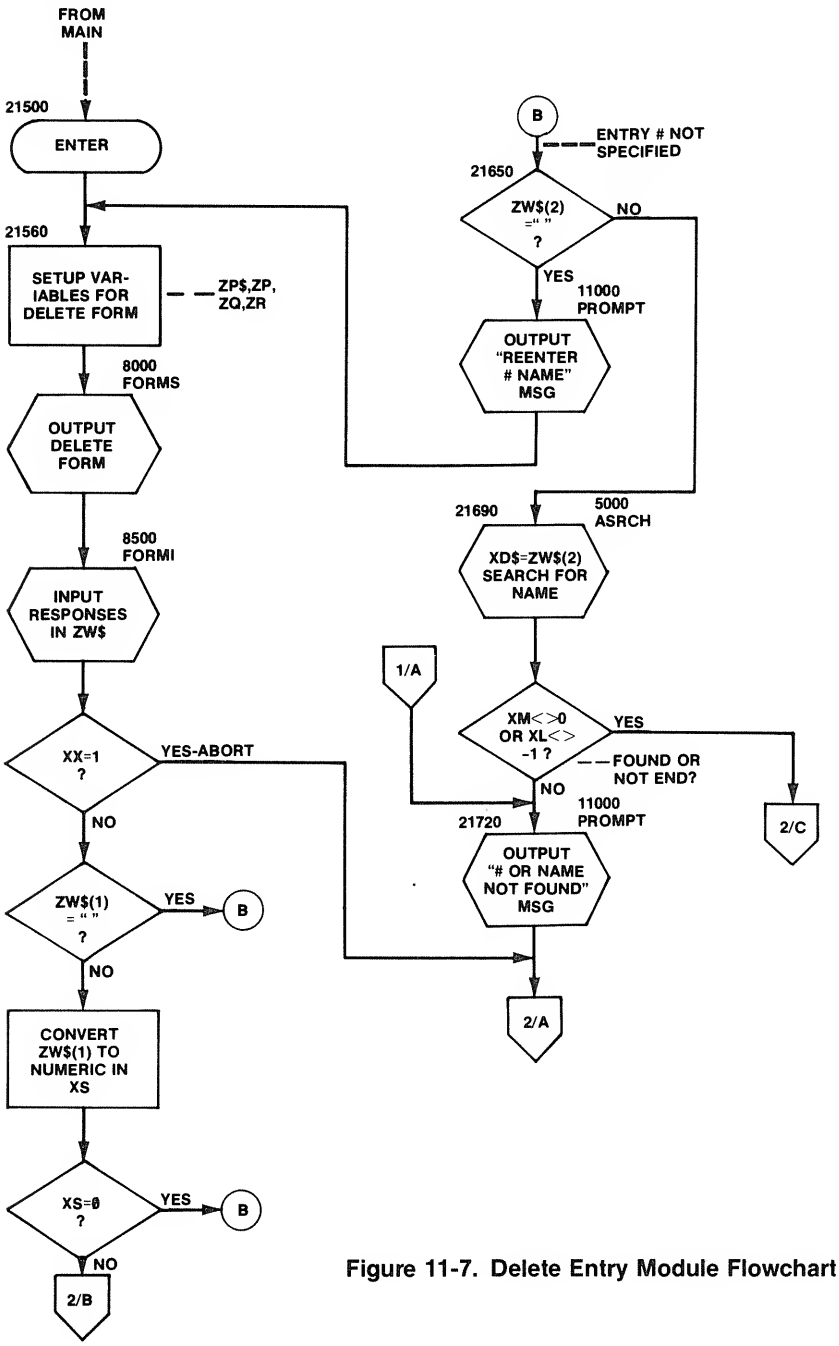


Figure 11-7. Delete Entry Module Flowchart

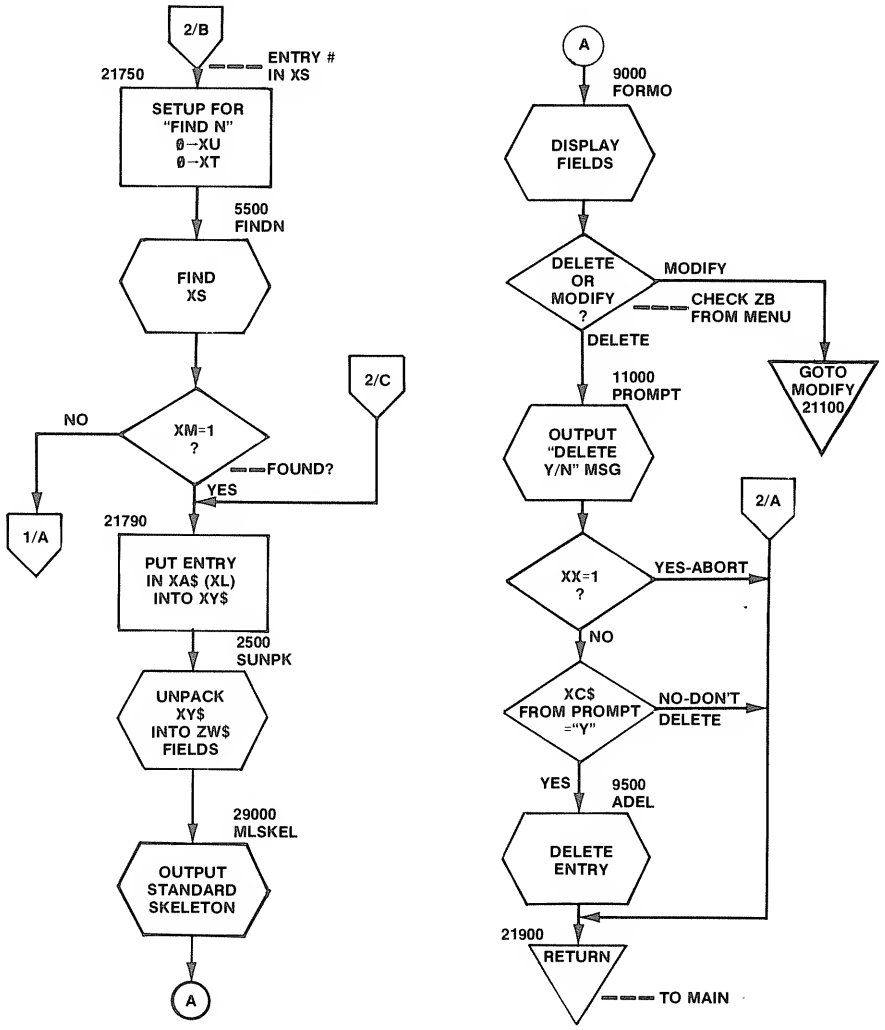


Figure 11-7. Delete Entry Module Flowchart (con't)

The actions that we have to take to delete an entry are as follows:

1. Input an entry number or entry name for the delete
2. Search for that number or name
3. If the entry is found, display the fields of the entry in a form and ask whether the entry is truly to be deleted.
4. If the entry is to be deleted, delete the entry from the list; if not, return to the menu.
5. Return to the main menu.

```
21500 GOTO 21550 'MFDEL
21510 '*****************************************************
21520 ' THIS IS DELETE ENTRY PROCESSING
21530 '*****************************************************
21540 'FIRST OUTPUT DELETE SKELETON
21550 ZP$(0)="DELETE ENTRY"
21560 ZP=55:Z0=2:ZP$(1)="ENTRY # (ENTER IF NOT KNOWN)"
21570 ZP$(2)="NAME1 STRING (ENTER IF NOT KNOWN)"
21580 ZR(1)=3:ZR(2)=15:GOSUB 8000
21590 'NOW READ IN FIELDS
21600 GOSUB 8500 : IF XX=1 GOTO 21900
21610 IF ZW$(1)="" GOTO 21650
21620 XS=VAL(ZW$(1))
21630 IF XS=0 GOTO 21650
21640 GOTO 21750
21650 IF ZW$(2)<>"" GOTO 21690
21660 XB=3:XB$="REENTER VALID # OR NAME":GOSUB 11000
21670 GOTO 21560
21680 'SEARCH FOR FIELD 1 NAME HERE
21690 XD$=ZW$(2):GOSUB 5000
21700 'IF FOUND, CONTINUE
21710 IF XM<>0 OR XL<>-1 GOTO 21790
21720 XB=3:XB$="# OR NAME NOT FOUND":GOSUB 11000
21730 GOTO 21900
21740 'NOW HAVE ENTRY # IN XS - FIND NTH ENTRY
21750 XU=0:XT=0:GOSUB 5500
21760 'IF FOUND CONTINUE
21770 IF XM=0 GOTO 21720
21780 'NOW GET ENTRY STRING AND "UNPACK" IT
21790 XY$=XA$(XL):GOSUB 2500
21800 'DISPLAY MAIL LIST SKELETON
21810 GOSUB 29000
21820 'DISPLAY FIELDS ON FORM
21830 GOSUB  9000
21840 'CHECK ZB FLAG FROM MENU SELECTION FOR DELETE OR MODIFY
21850 IF ZB=2 GOTO 21100
21860 XB$="DELETE YES OR NO":XB=1:GOSUB 11000:IF XX=1 GOTO 21900
21870 IF XC$="N" GOTO 21900
21880 'DELETE ENTRY
21890 GOSUB 9500
21900 RETURN
```

Figure 11-8. Delete Entry Module Listing

These are the general actions for the delete. There are some other actions that we also have to take. We must check for either an entry number or name; if neither has been input, then the user has made an error. If the entry is specified by number only, and that number is beyond the last number of the list, then we can't display or delete the item. Also, if the user wants to terminate the Delete operation, variable XX will be set to 1; we'll have to return to the menu in this case.

First of all we'll output the Delete "skeleton" form by setting up ZP$ and ZR and calling FORMS at line 8000. This form is shown in Figure 11-9. Next, we'll read in the fields for the form by calling FORMI at line 8500. FORMI stores the field input in the ZW$ array. On return from FORMI, variable XX may be set to 1 if the user

has terminated the delete operation at this point. If this is true, we'll RETURN from MFDEL without taking further action.

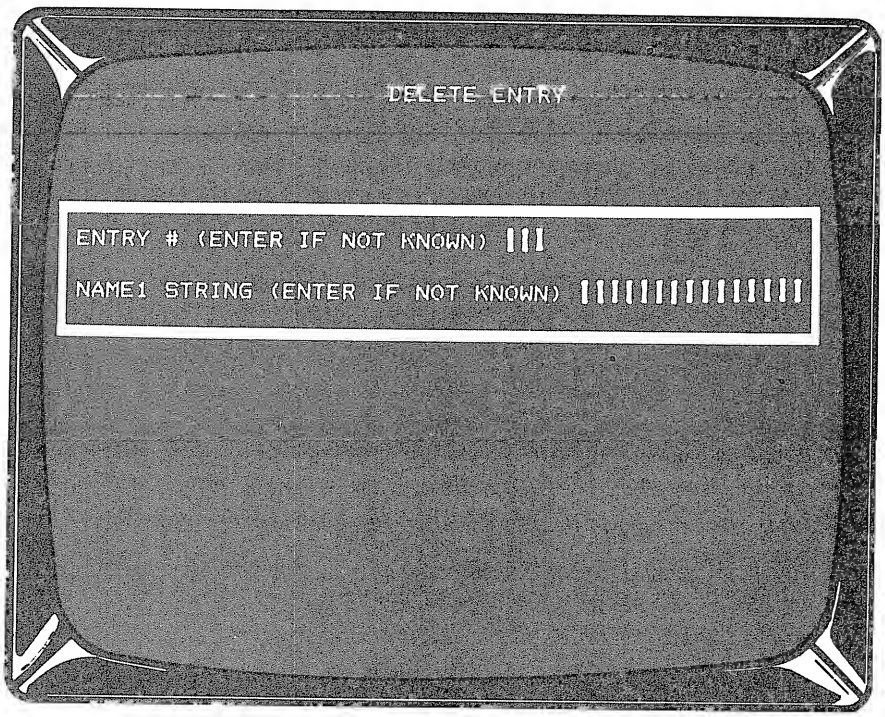


Figure 11-9. Delete Entry Form

At this point we have either an entry number or an entry name or both in ZW$(1) and ZW$(2). The code from line 21610 through 21740 tests the two fields. If neither has any input, the message REENTER VALID # OR NAME is displayed in the prompt message area and the Delete form is again output.

If an entry number is in ZW$(1), it **is converted to a numeric value** by XS=VAL(ZW$(1)), and line 21750 is executed. At line 21750, a call is made to FINDN at line 5500. FINDN finds the entry number defined by XS. If the number is out of range, line 21720 displays the # OR NAME NOT FOUND message and RETURNs from MFDEL.

If ZW$(1) is blank, then ZW$(2) contains an entry name. Delete must search for this name and find the corresponding entry number. It does so by setting XD$ equal to ZW$(2) and calling the ASRCH module. ASRCH searches the current list in memory and returns flag XM=1 if the name is found. If the end of the list is reached variable XL is set to -1. If XM<>0 OR XL<>-1, the entry has been

found and variable XS contains the entry number (from ASRCH); a branch is then made to line 21790. If this is not true, the message # OR NAME NOT FOUND is displayed by calling PROMPT at line 11000 and a RETURN is made from MFDEL.

At this point we're at line 21790 with variable XL containing the current entry location in XA$. This represents either the entry number defined in the ENTRY # field, or the entry containing the name defined in the NAME1 STRING field. In the latter case, this is the first entry containing the name.

The entry defined by XA$(XL) is now transferred to XY$ for "unpacking" in SUNPK (line 2500). The complete string in XY$ is unpacked into fields ZW$(1) through ZW$(8) after SUNPK.

Next, we'll retrieve the skeleton in the closet by MLSKEL at line 29000. The fields in ZW$ are then displayed in the skeleton by calling FORMO at line 9000. FORMO takes the fields in ZW$ and displays them at the proper places in the skeleton.

All that's left now is to ask the user whether he wants to delete the entry. The prompt message DELETE YES OR NO is displayed. If XC$ is "N," a RETURN is made without deleting. If the response is "Y" a delete entry call is made to the ADEL module at line 9500 to remove the entry from the list. ADEL uses variables XJ, XK, and XL to delete the entry from XA$ and changes the pointers in XA%.

The ZB flag action in line 21850 will be explained in the Modify Entry module, next. A typical delete entry action is shown in Figure 11-10.

STEP	VARIABLES
1. FIND STRING IF NOT # (ASRCH)	ZW$(2)→XD$, XJ, XK, XL, XS
2. FIND # (FINDN)	XS, XJ, XK, XL
3. GET ENTRY	XA$(XL)→XY$
4. UNPACK ENTRY (SUNPK) AND DISPLAY (FORMO)	XY$→ZW$ ARRAY
5. DELETE IF REQUESTED (ADEL)	XA$(XL) DELETED FROM XA$

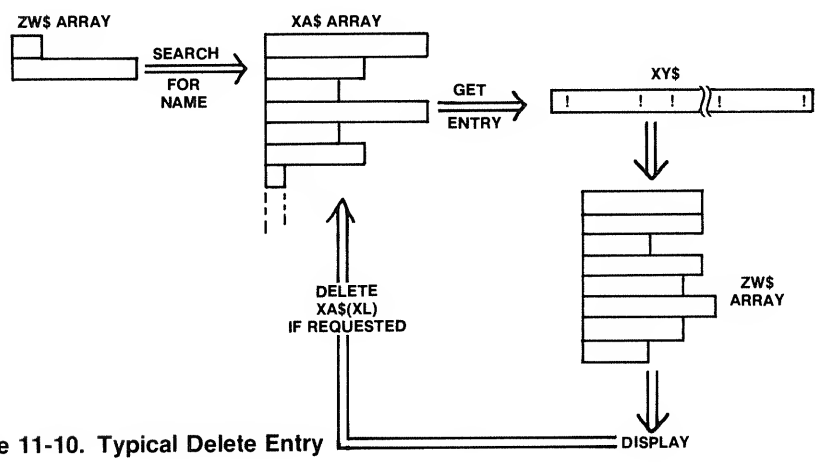


Figure 11-10. Typical Delete Entry

Modify Entry Processing — MFMOD

If we refer to the Design Spec we can see that Modify Entry is very similar to the actions of Delete Entry. A specified entry is located by number or name. The entry is then displayed on the standard mail list form. The user is queried as to

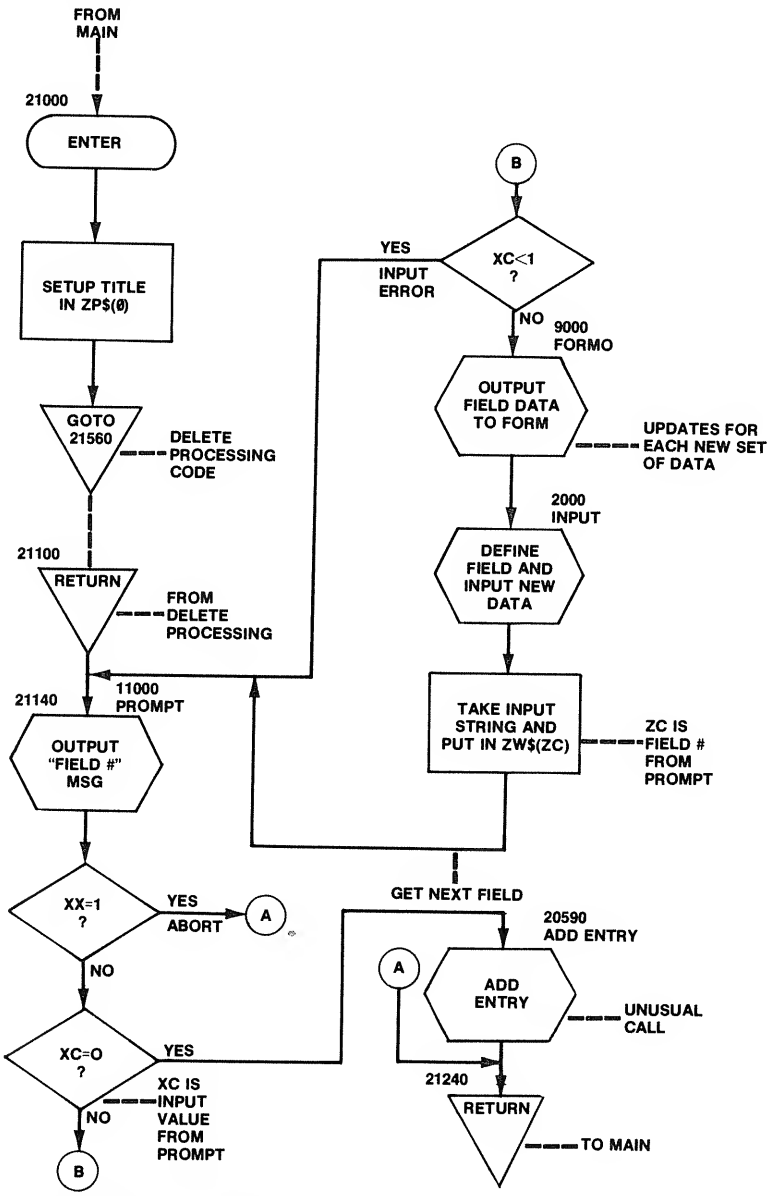


Figure 11-11. Modify Entry Module Flowchart

whether he truly wishes to modify the entry. If yes, the entry is deleted. All of the actions up to this point are identical to the Delete Entry action in MFDEL.

Modify Entry next asks the user which fields are to be modified and inputs the new fields. After the last field has been input, MFMOD adds the modified entry to the list.

The MFMOD flowchart is shown in Figure 11-11 and the actual code in Figure 11-12.

```
21000 GOTO 21050 'MFMOD
21010 '*************************************************************
21020 ' THIS IS MODIFY ENTRY PROCESSING
21030 '*************************************************************
21040 'FIRST OUTPUT MODIFY SKELETON
21050 ZP$(0)="MODIFY ENTRY"
21060 'GO TO DELETE PROCESSING. BOTH MODIFY AND DELETE HAVE VERY
21070 'SIMILAR PROCESSING AND IT CAN BE SHARED BETWEEN THE TWO!
21080 GOTO 21560
21090 'REENTER FROM DELETE PROCESSING HERE
21100 XB$="MODIFY YES OR NO":XB=1:GOSUB 11000:IF XX=1 GOTO 21240
21110 IF XC$="N" GOTO 21240
21120 'DELETE ENTRY HERE
21130 GOSUB 9500
21140    XB$="FIELD # TO MODIFY":XB=0:GOSUB 11000:IF XX=1 GOTO 21240
21150    IF XC=0 GOTO 21230
21160    IF XC<1 GOTO 21140
21170    GOSUB 9000
21180    'INPUT NEW STRING FOR FIELD
21190    ZC=ZS(XC):ZD=ZR(XC):ZE=1:GOSUB 2000 :IF XX=1 GOTO 21240
21200    ZW$(XC)=ZF$
21210    GOTO 21140
21220 'NOW "ADD" ENTRY BY USING ADD ENTRY PROCESSING!
21230 GOSUB 20590
21240 RETURN
```

Figure 11-12. Modify Entry Module Listing

The first action taken is to set the title in ZP$(0) to MODIFY ENTRY. A GOTO 21560 then enters the Delete Entry code. After the entry has been found by number or name and displayed, MFDEL checks the ZB variable from the menu selection (MENU). If ZB=2, the Modify Entry function is being processed, and control is returned back to line 21100 in MFMOD.

The entry has now been displayed on the screen. MFMOD next asks the question MODIFY YES OR NO. This gives the user an out if the wrong entry has been found. If he answers N to PROMPT (line 11000), the RETURN at line 21240 returns to the main menu. A RETURN is also made if variable XX is set to 1 from the user pressing the ENTER key (Models I/III) or "up arrow" key (Model II).

If the answer is Y, the ADEL module at line 9500 deletes the entry; variable XJ, XK, and XL are still set up from the search and are used in the delete. If we were to look at the XA$ list at this point, we would see that the entry displayed on the screen has been deleted. It only exists on the screen and in the ZW$ array (screen

fields) at this point. (If the CLEAR (Model I/III) or "up arrow" (Model II) is used now, the entry will disappear for good, unless you have total recall!)

The code from line 21140 through 21210 is the loop for allowing the user to modify any of the eight fields displayed on the screen. The message `FIELD # TO MODIFY` is first displayed in the prompt message area as shown in Figure 11-13. The user then enters a field number to `PROMPT`. `PROMPT` returns the response in XC. If XC is less than 1, the response is invalid, and line 21140 repeats the question. If the response is 0, ENTER has been pressed without a field number. In this case, the user has terminated the modify operation; line 21230 then takes the current field information in ZW$(1) through ZW$(8) and adds the entry. We could have written the add entry code here, but it exists in the Add Entry module, so a transfer to the proper Add Entry code is made by `GOSUB 20590`. After the Add Entry Code has added the entry to the list, a RETURN is made to line 21240 which in turn RETURNs to the main menu.

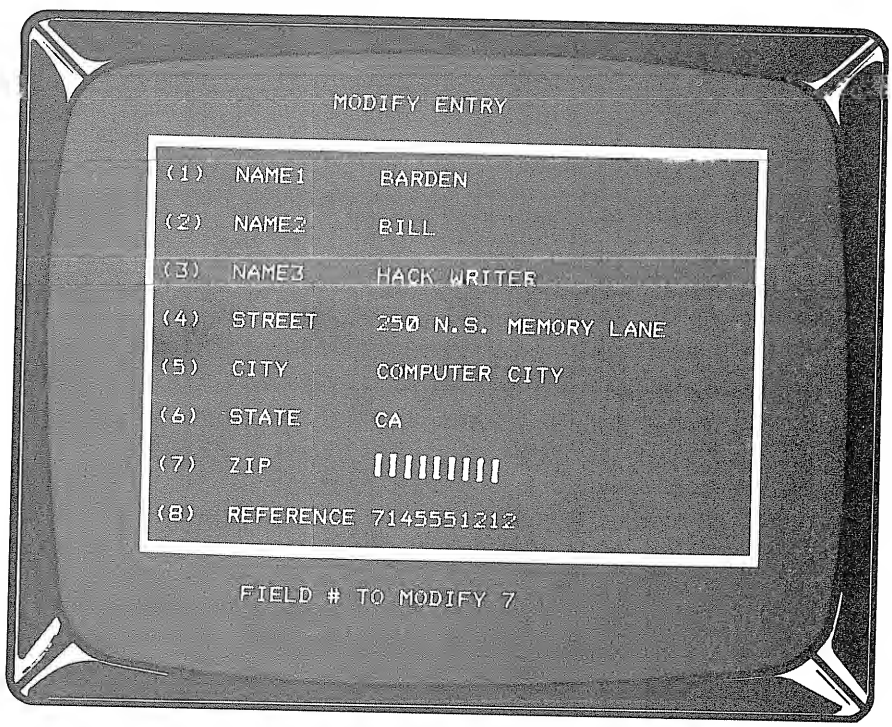


Figure 11-13. Modify Entry Prompting

If the user has entered a valid field number for the modify, the FORMO module is called at line 9000. The FORMO module takes the current field information in ZW$(1) through ZW$(8) and displays it on the form. This is necessary to "update" the screen from the last field modify.

Next, the field to be modified is erased by "fill" characters and new user input is started. This operation is shown in Figure 11-14. The INPUT module (line 2000) does this automatically. Before INPUT is called, the screen location of the field and maximum length of the field are picked up from the ZS and ZD arrays, respectively. These values were set by FORMS (form skeleton) when the MAIL-LIST form was displayed.

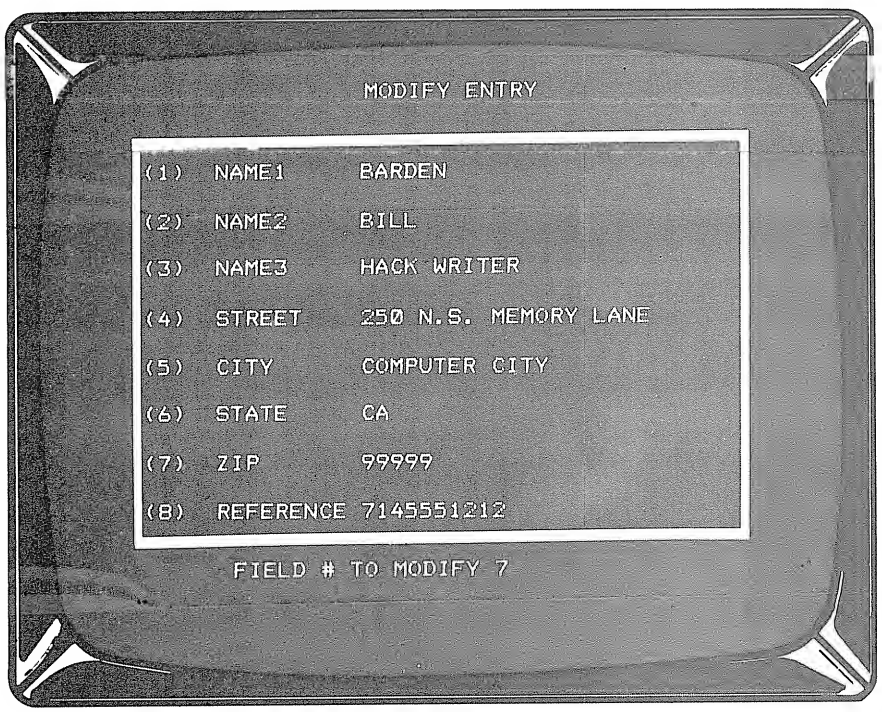


Figure 11-14. Modify Entry Field Entry

On RETURN from INPUT, ZF$ contains the new field input, and the field also appears on the screen. The proper field array ZW$(XC) is set to the new field, and a loop back to line 21140 is made for the next field to be modified. During this modify operation the user may press ENTER or "up arrow" to terminate the operation. If he does, a RETURN is made to the main menu. A typical modify entry action is shown in Figure 11-15.

STEP	VARIABLE
1. FIND STRING IF NOT # (ASRCH)	ZW$(2)→XD$, XJ, XK, XL, XS
2. FIND # (FINDN)	XS, XJ, XK, XL
3. GET ENTRY	XA$ (XL)→XY$
4. UNPACK ENTRY (SUNPK) AND DISPLAY (FORMO)	XY$→ZW$ ARRAY
5. DELETE ENTRY IF MOD-IFICATION (ADEL)	XA$ (XL) DELETED FROM XA$
6. INPUT NEW FIELDS	ZW$
7. GO TO ADD ENTRY PROCESSING	

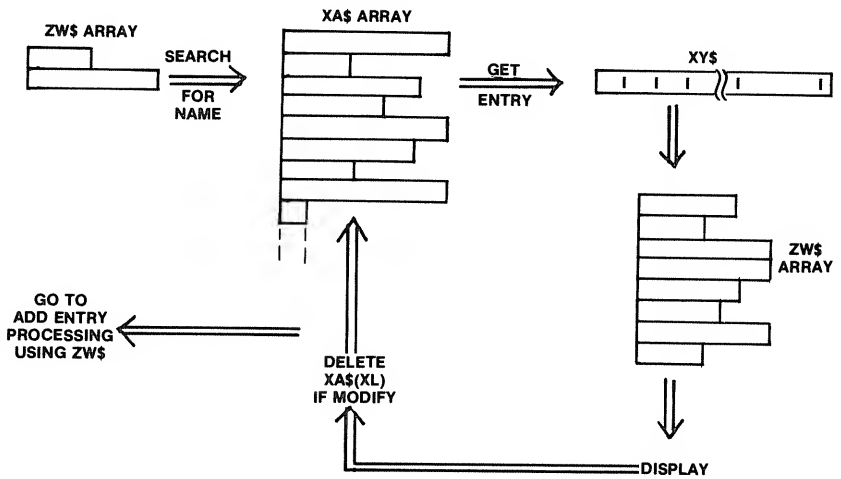


Figure 11-15. Typical Modify Entry

Chapter Twelve
MAILLIST—Displaying and Printing Entries

The MFDISP module is the largest module of MAILLIST. It is large because it combines the two functions of display of entries and printing of entries, in addition to subordinate functions such as defining print formats and handling display of secondary sorts. MFDISP could have been split up into two separate modules, one for display and one for printing. Despite the size, however, MFDISP actions are fairly straightforward and we'll describe them in detail so that you'll have no problems.

The flowchart for MFDISP is shown in Figure 12-1 while the actual code is shown in Figure 12-2.

Defining the Range

The code from line 22050 through line 22530 is concerned simply with defining the "range" of entries to be displayed or printed, in addition to specifying whether a "primary" or "secondary" sort is to be used.

The first action taken in this part of the code is to call FORMS (line 8000) with the one-entry form PRIMARY(P) OR SECONDARY(S) KEY?. After this form has been displayed by FORMS, FORMI is called at line 8500 to get the user input.

If the user input is not "P" or "S," the entire form is displayed once again, and a new FORMI occurs.

If ZW$(1) (the input from FORMI) is "P," the code at 22220 is executed, otherwise line 22120 is executed. The display/print for a primary sort and the display/-print for a secondary sort are somewhat different in their "range" definitions. The range of entries for a primary sort may be defined by a start number or by a start name and an end number or an end name. The range of entries for a secondary sort may only be defined by a start number and an end number. The reason for the limitation on the secondary sort is that it is much easier to find the "nth" entry in XB% (the secondary sort "pointers" array) then to find a name. (That's programmer lethargy again . . .)

If a secondary sort is specified, variable YT is checked. If YT is 0, the secondary array has never been sorted, or an old sort has been invalidated by a new entry, deletion, or modification. In this case the message NEVER SORTED OR MODIFIED-RESORT! is displayed by calling the PROMPT module at line 11000, and MFDISP starts again.

If YT is not 0, the secondary sort can proceed. First, the secondary sort form shown in Figure 12-3 is displayed by calling the FORMS module at line 8000. Next, the two fields associated with the form are read in by calling the FORMI

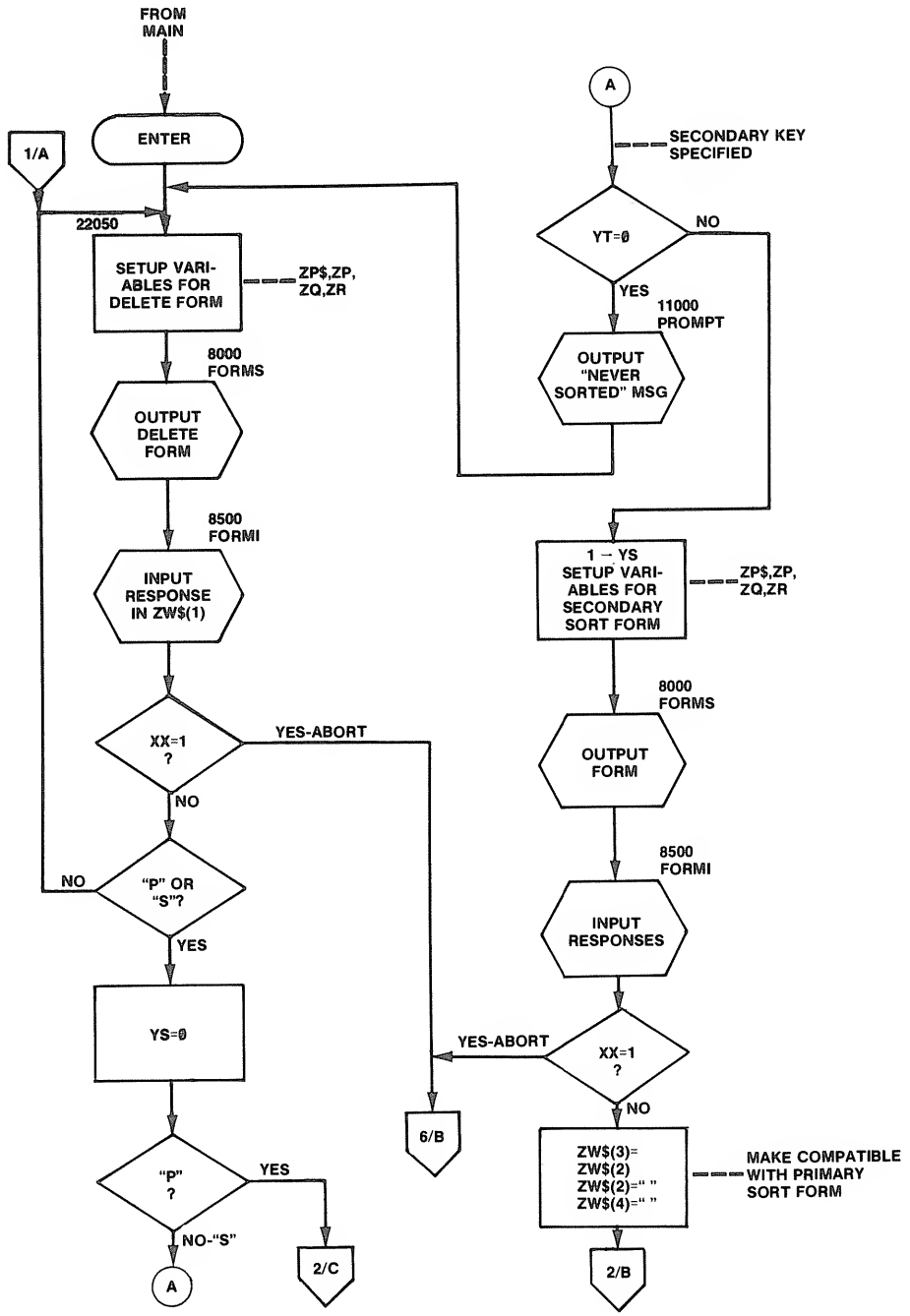


Figure 12-1. Print/Display Module Flowchart

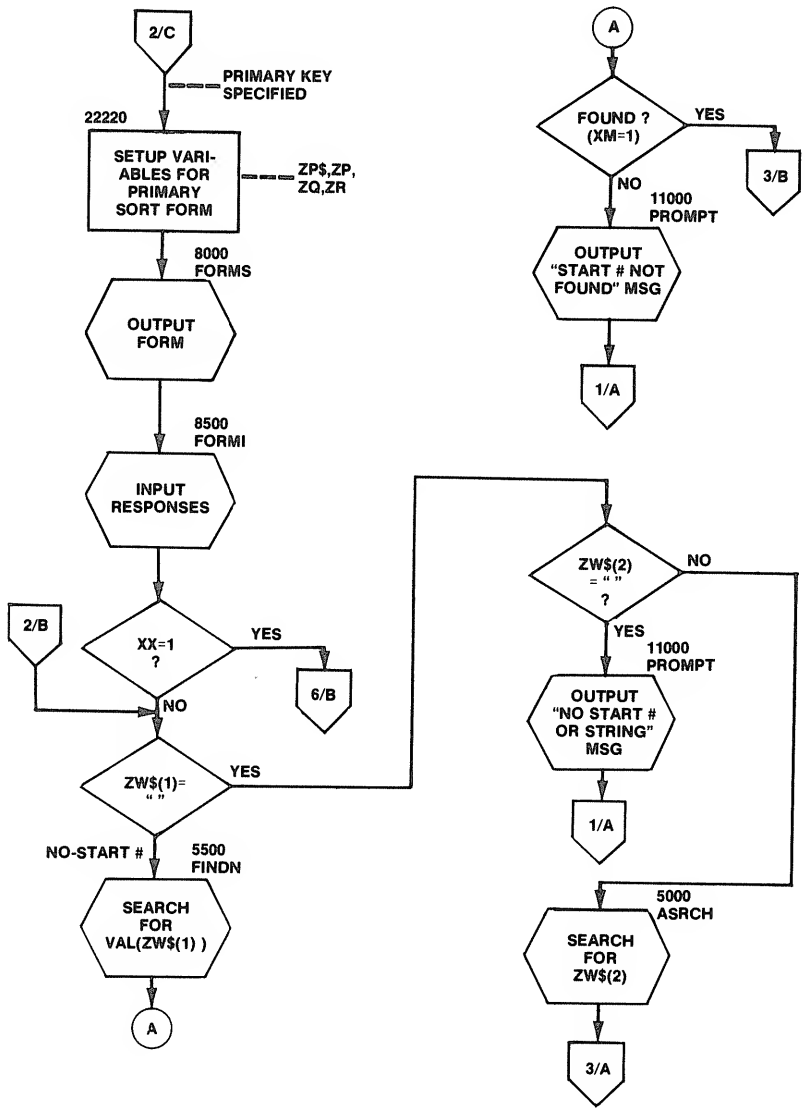


Figure 12-1. Print/Display Module Flowchart (cont.)

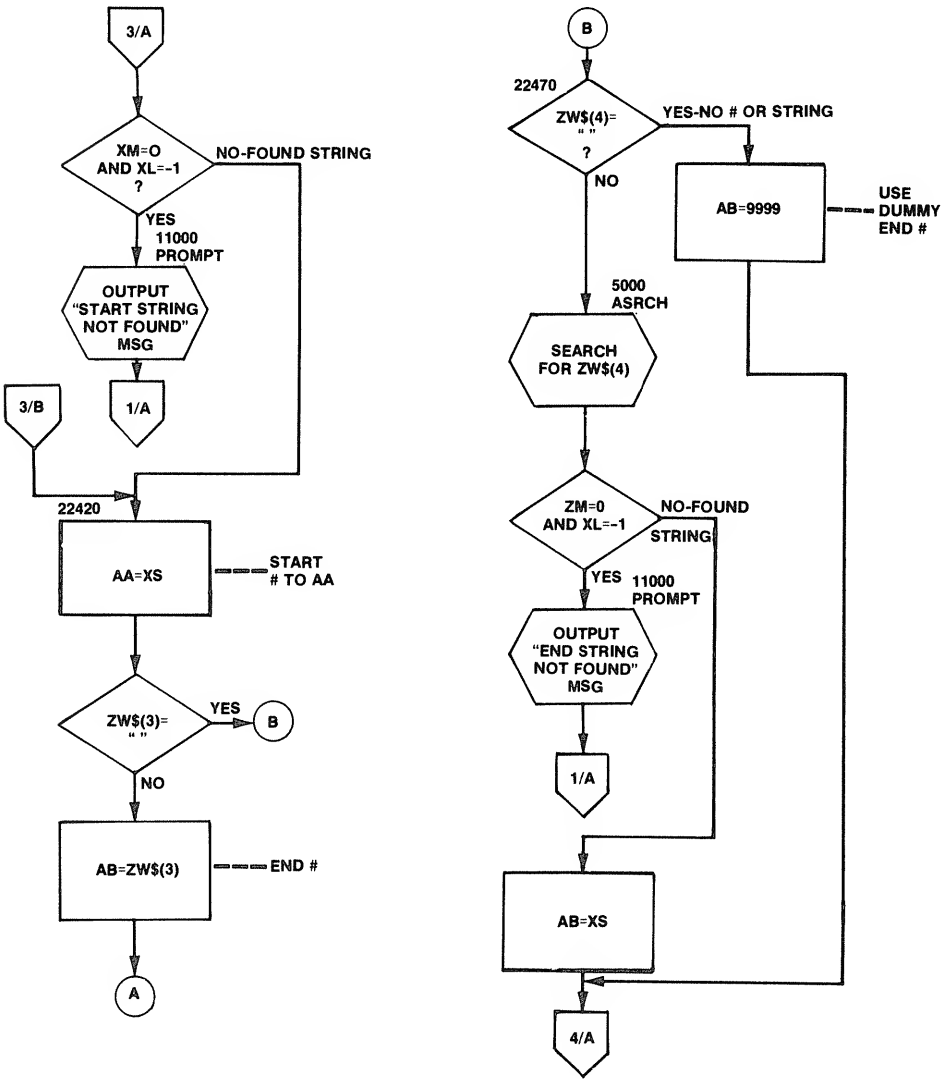


Figure 12-1. Print/Display Module Flowchart (cont.)

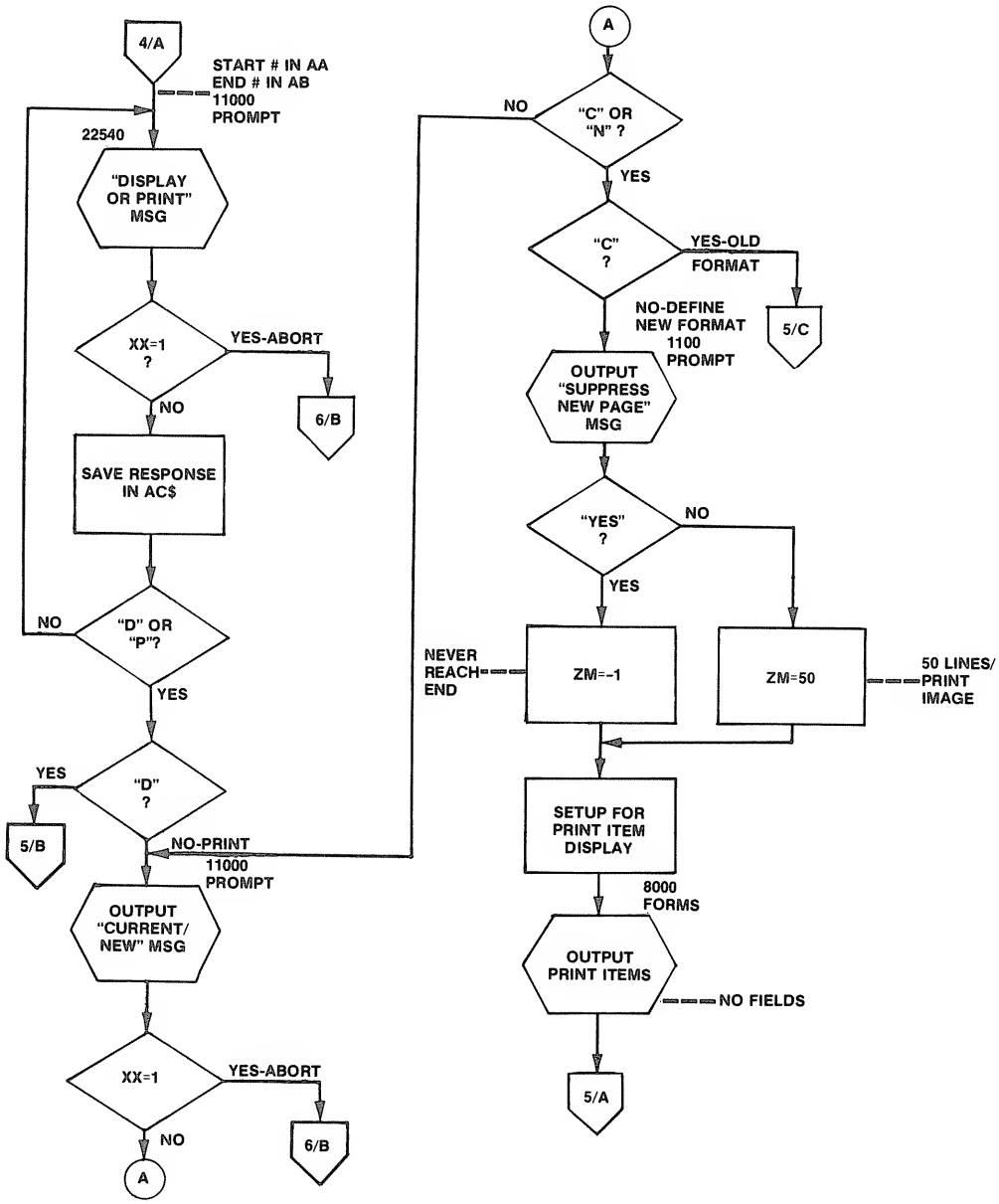


Figure 12-1. Print/Display Module Flowchart (cont.)

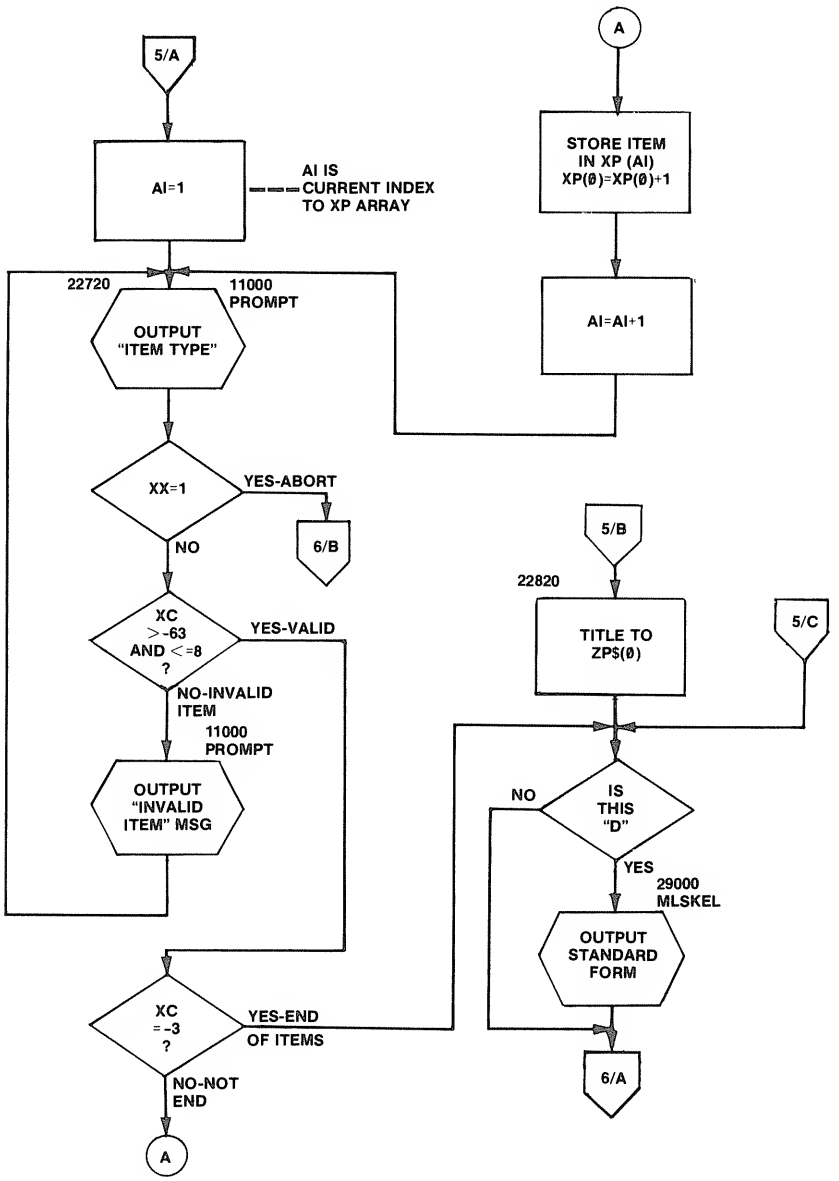


Figure 12-1. Print/Display Module Flowchart (cont.)

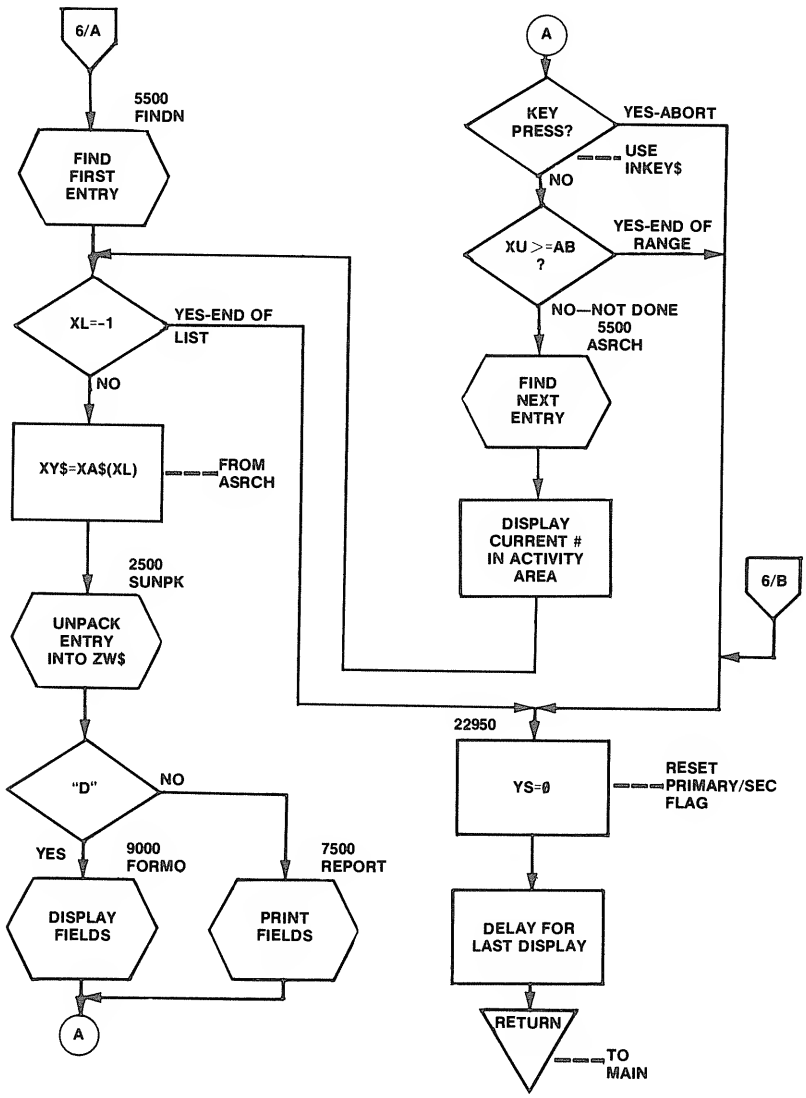


Figure 12-1. Print/Display Module Flowchart (cont.)

```
22000 GOTO 22050 'MFDISP
22010 '*************************************************************
22020 ' THIS IS DISPLAY/PRINT PROCESSING
22030 '*************************************************************
22040 'FIRST CHECK FOR PRIMARY OR SECONDARY KEY
22050 ZP=55:ZQ=1:ZP$(0)="DISPLAY/PRINT"
22060 ZP$(1)="PRIMARY (P) OR SECONDARY (S) KEY?"
22070 ZR(1)=1:GOSUB 8000
22080 GOSUB 8500 :IF XX=1 GOTO 22950
22090 IF ZW$(1)<>"P" AND ZW$(1)<>"S" GOTO 22050
22100 YS=0:IF ZW$(1)="P" GOTO 22220
22110 'SECONDARY KEY HERE
22120 IF YT<>0 GOTO 22150
22130 XB=3:XB$="NEVER SORTED OR MODIFIED-RESORT!":GOSUB 11000
22140 GOTO 22050
22150 YS=1:ZP=55:ZQ=2:ZP$(0)="SECONDARY DISPLAY/PRINT"
22160 ZP$(1)="START # (ENTER IF NOT KNOWN)":ZP$(2)="END # (ENTER IF NOT KNOWN)"
22170 ZR(1)=3:ZR(2)=3:GOSUB 8000
22180 'NOW READ IN FIELDS AND REARRANGE FOR CHECK
22190 GOSUB 8500 :IF XX=1 GOTO 22950
22200 ZW$(3)=ZW$(2):ZW$(2)="":ZW$(4)=""
22210 GOTO 22310
22220 ZP=55:ZQ=4:ZP$(0)="PRIMARY DISPLAY/PRINT"
22230 ZP$(1)="START# (ENTER IF NOT KNOWN)"
22240 ZP$(2)="START ENTRY (ENTER IF NOT KNOWN)"
22250 ZP$(3)="END# (ENTER IF NOT KNOWN)"
22260 ZP$(4)="END ENTRY (ENTER IF NOT KNOWN"
22270 ZR(1)=3:ZR(2)=15:ZR(3)=3:ZR(4)=15:GOSUB 8000
22280 'NOW READ IN FIELDS
22290 GOSUB 8500 :IF XX=1 GOTO 22950
22300 'NOW CHECK START # OR STRING
22310 IF ZW$(1)="" GOTO 22370
22320 XU=0:XS=VAL(ZW$(1)):XT=0:GOSUB 5500
22330 IF XM<>0 GOTO 22420
22340 XB$="START # NOT FOUND"
22350 XB=3:GOSUB 11000
22360 GOTO 22050
22370 IF ZW$(2)<>"" GOTO 22390
22380 XB$="NO START # OR STRING":GOTO 22350
22390 XD$=ZW$(2):GOSUB 5000
22400 IF XM=0 AND XL=-1 THEN GOTO 22410ELSE GOTO 22420
22410 XB$="START STRING NOT FOUND":GOTO 22350
22420 AA=XS
22430 'NOW FIND END # OR STRING
22440 IF ZW$(3)="" GOTO 22470
22450 AB=VAL(ZW$(3))
22460 GOTO 22540
22470 IF ZW$(4)<>"" THEN GOTO 22500
22480 AB=9999
22490 GOTO 22540
22500 XD$=ZW$(4):GOSUB 5000
22510 IF XM<>0 OR XL<>-1 GOTO 22530
22520 XB$="END STRING NOT FOUND":GOTO 22350
22530 AB=XS
22540 XB$="DISPLAY(D) OR PRINT(P)?":XB=1:GOSUB 11000:IF XX=1 GOTO 22950
22550 AC$=XC$
22560 IF AC$<>"D" AND AC$<>"P" GOTO  22540
22570 IF AC$="D" GOTO 22820
```

Figure 12-2. Print/Display Module Listing

```
22580 'PRINT HERE
22590 XB$="CURRENT FORMAT (C) OR NEW (N)?":XB=1:GOSUB 11000:IF XX=1 GOTO 22950
22600 IF XC$<>"C" AND XC$<>"N" GOTO 22590
22610 IF XC$="C" GOTO 22830
22620 'DEFINE NEW FORMAT HERE
22630 XB$="SUPPRESS NEW PAGE, Y OR N?":XB=2:GOSUB 11000
22640 IF XC$="Y" THEN ZM=-1 ELSE ZM=50
22650 ZP=55:ZQ=6:ZP$(0)="DISPLAY/PRINT FORMAT ITEMS"
22660 ZP$(1)="0=NEW LINE":ZP$(2)="1-N=FIELD N"
22670 ZP$(3)="-M=TAB TO POSITION M":ZP$(4)="-1=NEW PAGE"
22680 ZP$(5)="-2=PRINT REPORT COUNTER XN":ZP$(6)="-3=END ITEM DEFINITION"
22690 ZR(1)=0:ZR(2)=0:ZR(3)=0:ZR(4)=0:ZR(5)=0:ZR(6)=0:GOSUB 8000
22700 XP(0)=0
22710 AI=1
22720    XB$="ITEM TYPE?:":XB=0:GOSUB 11000:IF XX=1 GOTO 22950
22730    IF XC>-63 AND XC<=8 GOTO 22760
22740    XB$="INVALID ITEM TYPE - IGNORED":XB=3:GOSUB 11000
22750    GOTO 22720
22760    IF XC=-3 GOTO 22830
22770    XP(AI)=XC
22780    XP(0)=XP(0)+1
22790    AI=AI+1
22800    GOTO 22720
22810 'DISPLAY OR PRINT HERE
22820 ZP$(0)="DISPLAY ENTRY"
22830 IF AC$="D" THEN GOSUB 29000
22840 XU=0:XS=AA:XT=0:GOSUB 5500
22850 XS=-1:XT=1
22860    IF XL=-1 GOTO 22950
22870    XY$=XA$(XL):GOSUB 2500
22880    XN=XU-1
22890    IF AC$="D" THEN GOSUB 9000 ELSE GOSUB 7500
22900    IF INKEY$<>"" GOTO 22950
22910    IF XU>=AB GOTO 22950
22920    GOSUB 5500
22930    PRINT @ YL, XU-2;"    ";
22940    GOTO 22860
22950 YS=0
22960    FOR AI=1 TO 900:NEXT AI
22970 RETURN
```

Figure 12-2. Print/Display Module Listing (cont.)

module. A RETURN is made with ZW$(1) and ZW$(2) containing the input. After the input, variable XX may be set to a 1 if the user wanted to terminate the operation. In this case a RETURN is made by a GOTO 22950. The next step is to check the start and end numbers. As this is a similar condition to the primary sort, the common check at line 22310 is executed after first setting ZW$(3) equal to ZW$(2) and ZW$(4)=ZW$(2)=" " (null string).

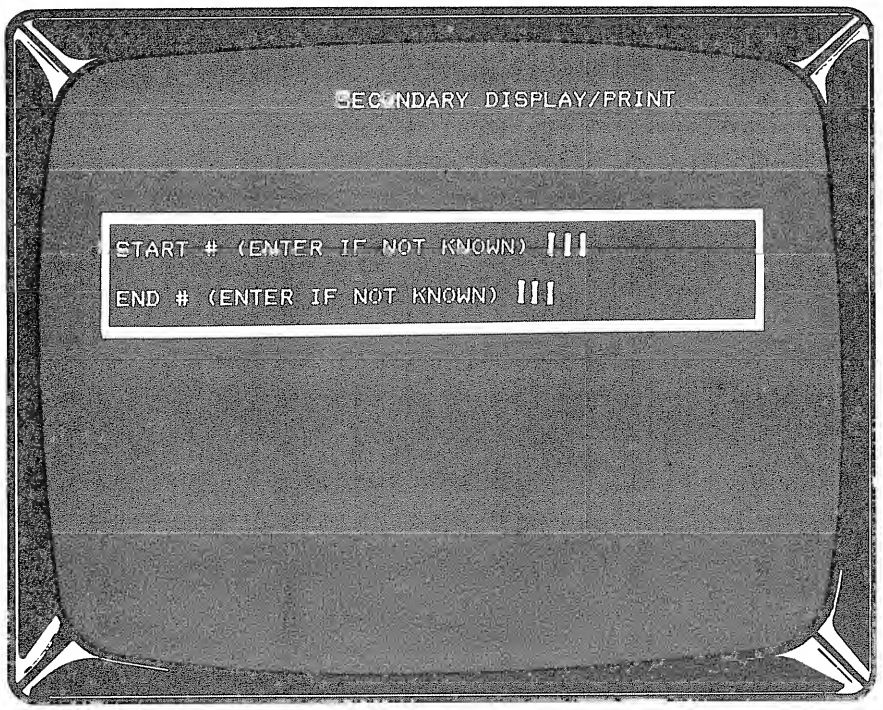


Figure 12-3. Secondary Sort Form

If a primary sort is called for the code starting at line 22220 is entered. This displays the form shown in Figure 12-4 by a call to the FORMS module at line 8000. The four fields are now read in by a call to FORMI (line 8500).

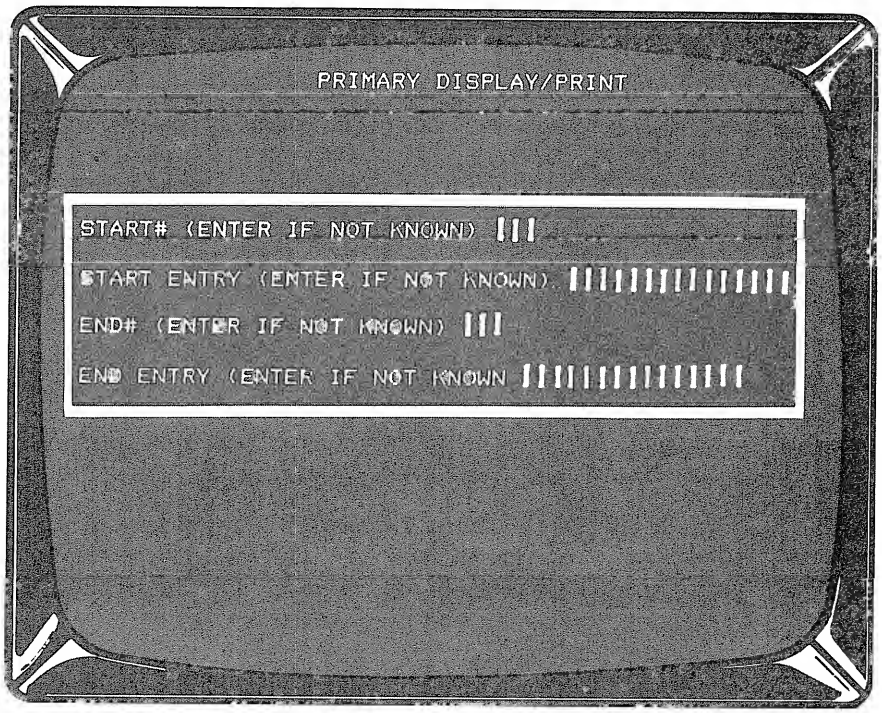


Figure 12-4. Primary Sort Form

Line 22310 is executed for both the primary and secondary sort cases. The code from line 22310 through 22530 is concerned with finding the start and end entry number for the range, and converting a name to an entry number if necessary.

If there is neither a start number in ZW$(1) nor a name in ZW$(2), then the message NO START # OR STRING is displayed by a call to the PROMPT module (line 11000).

If there is a start number in ZW$(1), then a possible name in ZW$(2) is ignored. A call is made to the FINDN module to look for XS=VAL(ZW$(1)). If this start number is not found, the message START NUMBER NOT FOUND is displayed by a call to PROMPT, and MFDISP is started over. (The only way for the start number not to be found is if the number of entries in the list exceed the start number.) If the start number is found, AA is set equal to XS in line 22420.

If there is no start number in ZW$(1) but a name in ZW$(2), line 22390 is executed. This line calls the ASRCH module to look for the name, and in the process, find the entry number in variable XS. If the name is not found, the

message START STRING NOT FOUND is displayed by a call to PROMPT, and the MFDISP module is restarted. If the name is found, variable AA is set equal to XS in line 22420.

We now have the start number in AA in line 22420. We must now go through a similar process for the end number or name. If there is neither end number in ZW$(3) nor name in ZW$(4), variable AB is set equal to 9999 in line 22480, and line 22540 is executed. This case allows the user to display/print the entire list from a given point. If there is a start number, a possible name is ignored and variable AB is set equal to VAL(ZW$(3)). If there is no end number but an end name, XD$ is set equal to ZW$(4) and a call is made to the ASRCH module (line 5000) at line 22500 to find the entry number in XS corresponding to the name. If the name is not found, the message END STRING NOT FOUND is displayed and MFDISP is restarted. If the name is found, variable AB is set equal to the entry number XS containing the name and line 22540 is executed.

Defining the Print Format

At this point, line 22540, we have the start entry number in AA and the end entry number in AB. I know, it was a lot of work just to get those two numbers, but sometimes code does become somewhat messy to accomplish trivial results (and quite often it's the reverse — trivial code for messy results . . .)

Next, a form containing one field is output — DISPLAY(D) OR PRINT(P)?. Variable AC$ is set equal to the input of XC$. Variable XC$ will be used later in the Display/Print procedure. If AC$ is neither "D" nor "P," the form is again displayed at line 22540. If variable XX is equal to 1, the user wants to terminate the operation and a RETURN is made by a GOTO 22950.

If a D has been input, a branch is made to line 22820. The remainder of the code from 22590 through 22800 is concerned with defining the print format.

The next form to be output is CURRENT FORMAT (C) OR NEW (N)?. If the input string XC$ is neither C nor N, the operation is repeated at line 22590. If C is specified, the current print format will be used, and a GOTO 22830 is executed.

If N is specified, a new print format will be defined in the code from line 22630 through 22800. This print format will consist of a series of print format items held in array XP. They will replace the last set of print format items held in the array and define a user-tailored print format.

Before the format is defined, the message SUPPRESS NEW PAGE, Y OR N? is displayed by the PROMPT module. If the response in XC$ is "Y" then ZM, the number of lines per page, is set to -1, otherwise the number of lines per page is set to 50. The new page suppress is used for label printing and other continuous line printing where page separations are not desired.

Next, FORMS is called to display a form defining the possible display/print format items (Figure 12-5). This is not really a form, as the form input fields are defined as zero length; it is more descriptive text.

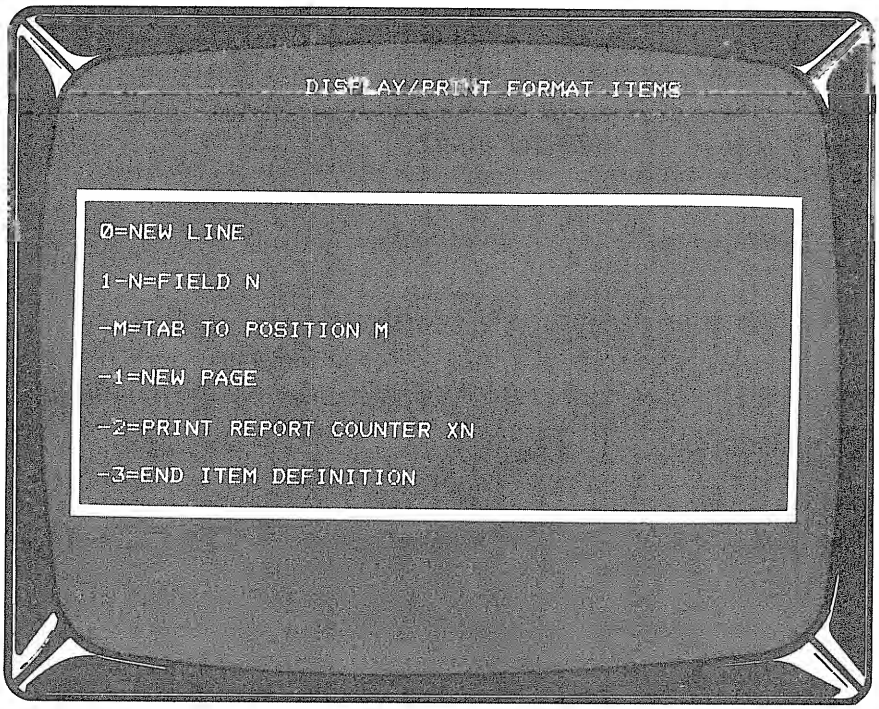


Figure 12-5. Print Format Item Form

The code from line 22700 through 22800 defines the format item input loop. Each time through the loop the prompt message ITEM TYPE? is printed by a call to PROMPT. The response is RETURNed in XC as a numeric value (XB=0 on the call). If the item type input is greater than -63 and less than 9, it is entered in the XP array as a format item to be used in the REPORT module. If the item value is not in this range the message INVALID ITEM TYPE - IGNORED is displayed and the item type message is displayed again for new input. A special case occurs when the item type value is -3; -3 terminates the input. For each item type, XP(0) is incremented by one count.

Displaying and Printing the Range of Items

The code from line 22820 through 22940 either displays the entries over the range defined by entry number AA through AB or prints the entries over the same range. AC$ still contains the "P" or "D" response to the DISPLAY OR PRINT question.

If a display is to be done, MLSKEL (line 29000) is called to display the mailing list skeleton. Next, FINDN (line 5500) is called to find the starting entry number in AA. Now XS is set to -1 (find all entries) and XT is set to 1 (find next entry) for the FINDN calls in the following loop.

The loop from 22860 through 22940 displays or prints every entry over the range. First a check is made of the next entry number in XL. If it is -1, then the end of the list has been reached and line 22950 is executed to end MFDISP. If XL is not -1, XY$ is set equal to the entry string from XA$. The SUNPK module is then called (line 2500) to unpack XY$ into fields, with the fields going into the ZW$ array.

XN, the number of the entry for the REPORT module, is set to XU-1 each time through the loop. XU is set to the number of the next entry from FINDN. If a display is being done, FORMO (line 9000) is called to display the fields in ZW$, otherwise REPORT is called (line 7500) to print the fields as defined by the current print format items in the XP array.

After the display or print of the current item a check is made of INKEY$. INKEY$ will be a "non-null" if a key has been pressed. If INKEY$ does not equal "" the display/print is terminated. This is necessary to give the user an "out" if he wants to recover from printing a long list of entries. If XU is equal to or greater than AB, the end entry number, the display/print is also terminated by a GOTO 22950. If neither case is present, FINDN is called (line 5000) to find the next entry. The PRINT YL prints the number of the current entry in the "activity area" so that the user may use it for a reference number in other MAILLIST functions. A loop is then made back to line 22860 for a display/ print of the next entry.

After the last entry has been printed, YS, the variable that defines whether a primary or secondary sort is being displayed or printed, is set to 0, for primary. A short time delay is then entered in line 22960 so that the last entry does not disappear too quickly from the screen as the menu is displayed by the RETURN.

Chapter Thirteen
MAILLIST—Cassette/Disk and Auxiliary Functions

In this chapter we'll talk about the remaining MAILLIST functions — cassette and disk storage, search processing, and secondary sort processing. The actual application code for these functions make extensive use of the General Purpose Modules, and the programs here are rather short.

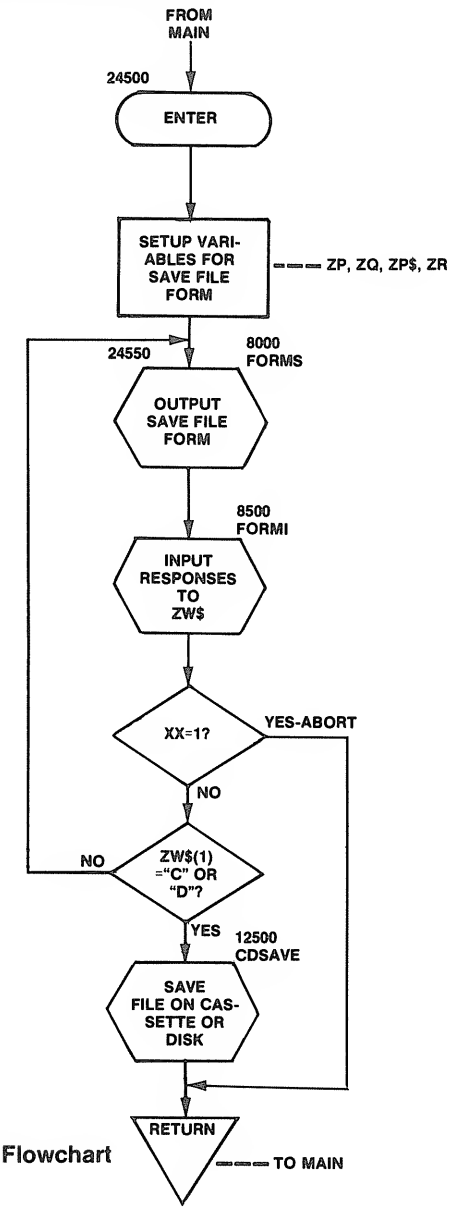


Figure 13-1. Save File Module Flowchart

Save File Processing

The flowchart for the MFSAVE module is shown in Figure 13-1 and the actual code is shown in Figure 13-2. The bulk of the code for MFSAVE is done in the CDSAVE module, located starting at line 12500.

```
24500 GOTO 24550 'MFSAVE
24510 '*************************************************************
24520 ' THIS IS SAVE FILE PROCESSING
24530 '*************************************************************
24540 'FIRST OUTPUT SAVE SKELETON
24550 ZP=55:ZQ=2:ZP$(0)="SAVE FILE"
24560 ZP$(1)="SAVE ON CASSETTE (C) OR DISK (D)?"
24570 ZP$(2)="DISK FILENAME?":ZR(1)=1:ZR(2)=15:GOSUB 8000
24580 'NOW READ IN RESPONSES
24590 GOSUB 8500 : IF XX=1 GOTO 24630
24600 IF ZW$(1)<>"C" AND ZW$(1)<>"D" GOTO 24550
24610 'SAVE TO CASSETTE OR DISK
24620 GOSUB 12500
24630 RETURN
```

Figure 13-2. Save File Module Listing

Save File is called from the menu by specifying menu item 8. It may be called at any time to save the current MAILLIST file in memory as a disk or cassette (Model I/III) file.

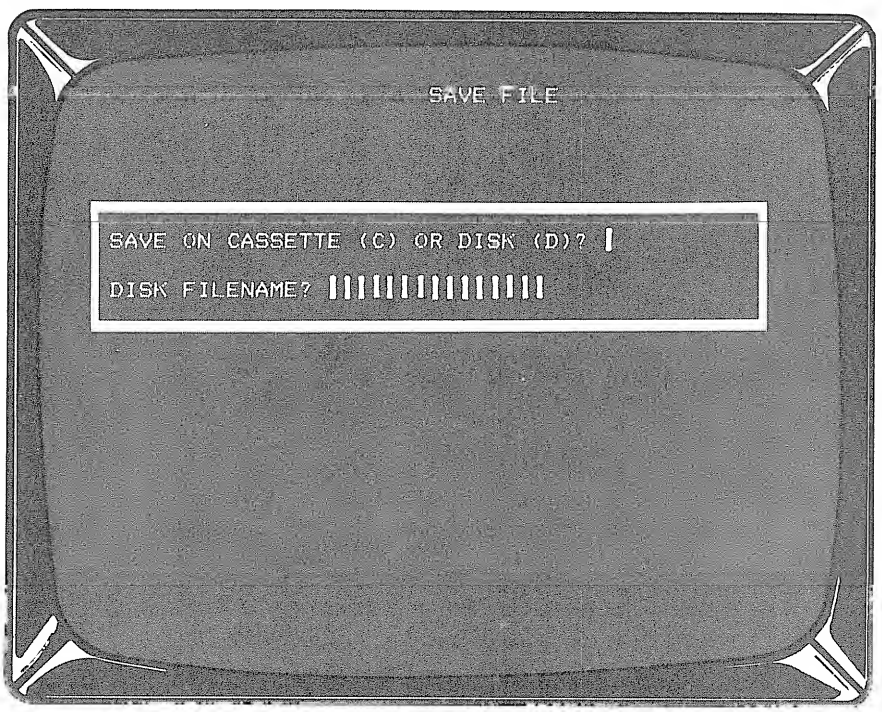


Figure 13-3. Save File Form

First, the Save File form is displayed on the screen by a call to the FORMS module (line 8000). This form is shown in Figure 13-3. The responses to the form questions are returned in ZW$(1) through ZW$(2) after a call to the FORMI module (line 8500). At this point ZW$(1) contains a C for cassette, or a D for disk, and ZW$(2) contains a disk file name (ZW$(2) is ignored for cassette).

The last action that MFSAVE takes before the RETURN is to call the CDSAVE module to write out the entire XA$ file to cassette or disk. The actions of CDSAVE are discussed in the previous section. Typical save file action is shown in Figure 13-4.

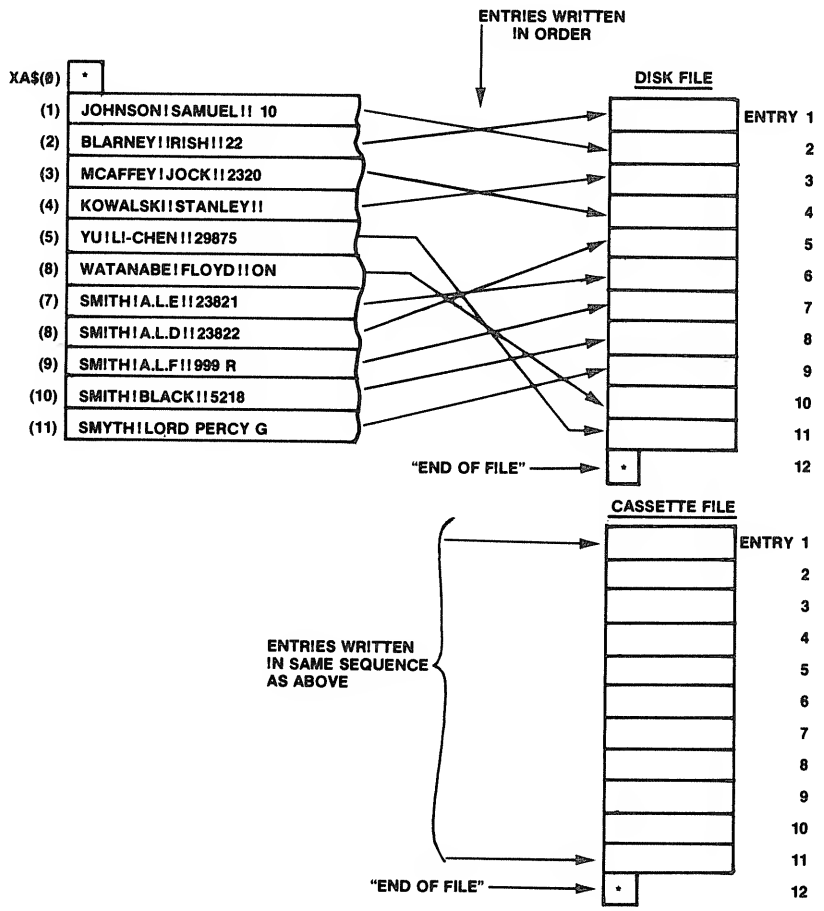


Figure 13-4. Typical Save File Action

Load File Processing

The flowchart for Load File is shown in Figure 13-5 while the actual code is shown in Figure 13-6. As in MFSAVE, most of the work is done in a General Purpose Module CDLOAD located starting at line 12000.

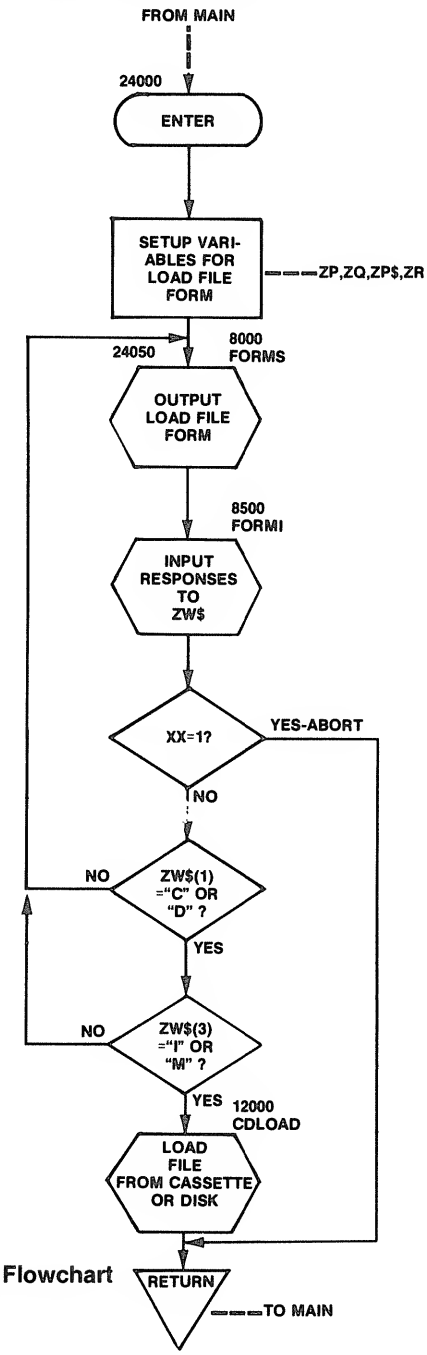


Figure 13-5. Load File Module Flowchart

```
24000 GOTO 24050 'MFLOAD
24010 '*****************************************************
24020 ' THIS IS LOAD FILE PROCESSING
24030 '*****************************************************
24040 'FIRST OUTPUT LOAD SKELETON
24050 ZP=55:Z0=3:ZP$(0)="LOAD FILE"
24060 ZP$(1)="LOAD FROM CASSETTE (C) OR DISK (D)?"
24070 ZP$(2)="DISK FILENAME?":ZP$(3)="INITIALIZE (I) OR MERGE (M)?"
24080 ZR(1)=1:ZR(2)=15:ZR(3)=1:GOSUB 8000
24090 'NOW READ IN RESPONSES
24100 GOSUB 8500 : IF XX=1 GOTO 24150
24110 IF ZW$(1)<>"C" AND ZW$(1)<>"D" GOTO 24050
24120 IF ZW$(3)<>"I" AND ZW$(3)<>"M" GOTO 24050
24130 'NOW READ IN FROM CASSETTE OR DISK
24140 GOSUB 12000
24150 RETURN
```

Figure 13-6. Load File Module Listing

MFLOAD is entered from a menu selection of 7. First, MFLOAD displays the Load File form on the screen by a call to FORMS. The Load File form is shown in Figure 13-7. The responses to the form questions are read into ZW$(1) through ZW$(3) by a call to the FORMI module. ZW$(1) contains a C for cassette or a D for disk, ZW$(2) contains a file name for disk, and ZW$(3) contains an I for initialize or an M for merge.

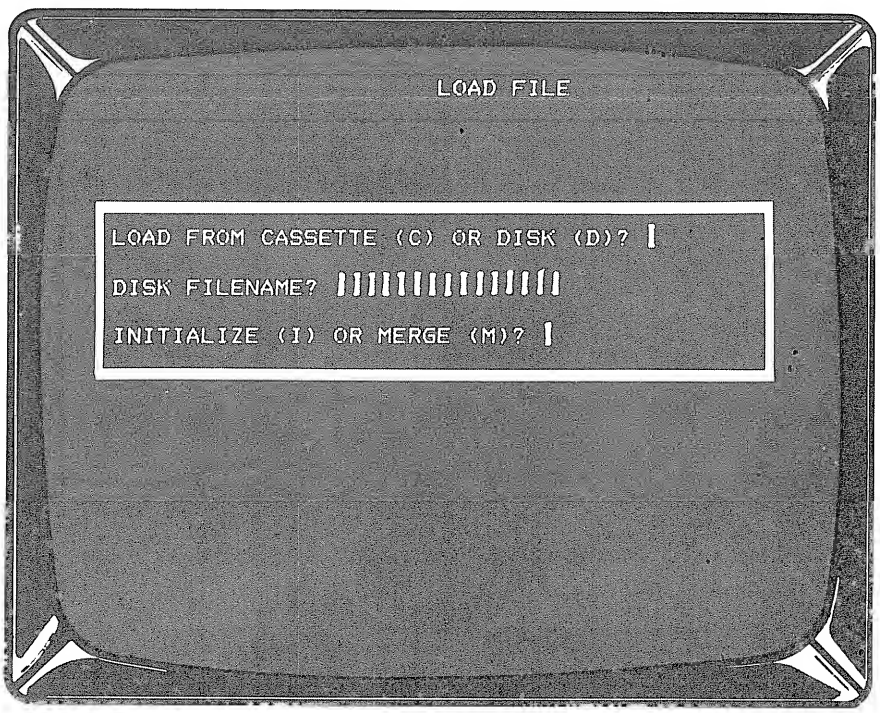


Figure 13-7. Load File Form

A check is made on the responses and the form is displayed again for proper input if the responses in ZW$(1) or ZW$(3) are incorrect.

CDLOAD is called by a GOSUB 12000. CDLOAD will read in the entire GPM-format file from disk into XA$ and adjust the pointers in XA%. If Initialize is specified, the entries from the file will be loaded in consecutive fashion; this is the "high-speed" mode. If Merge is specified, the entries from the disk file will be merged with the entries already in memory; this mode is slower but (as they say) powerful as it allows two or more files to be merged. The complete actions of CDLOAD are discussed in the previous section. Typical load file actions are shown in Figure 13-8.

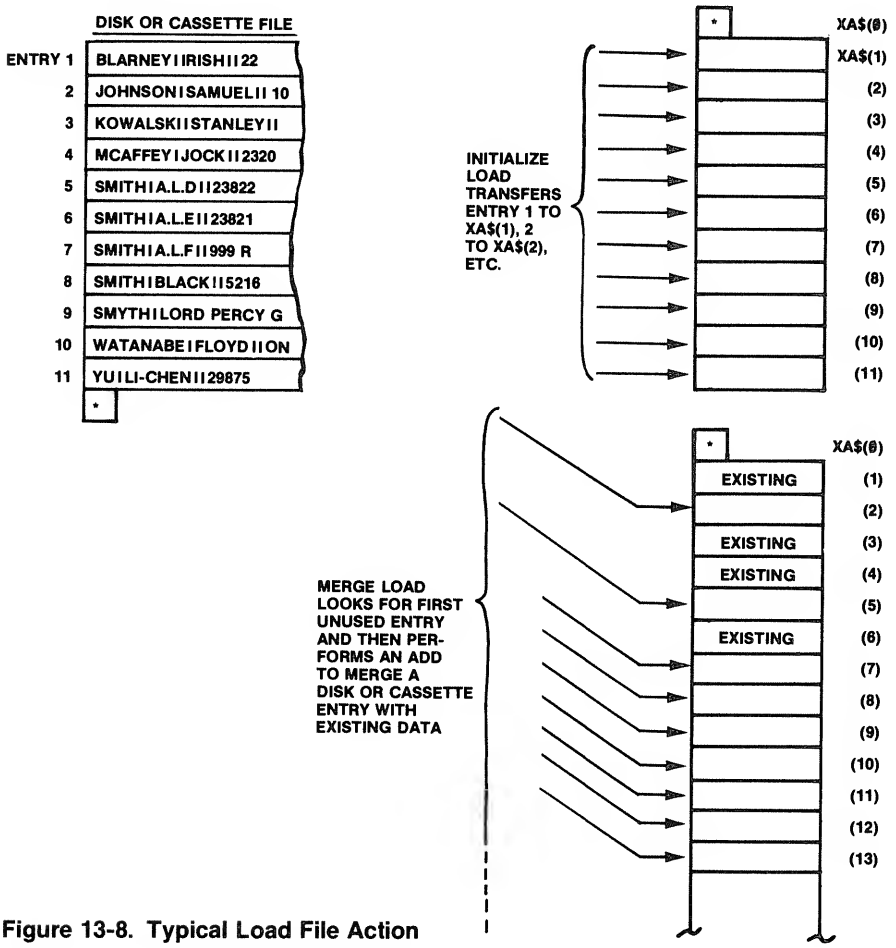


Figure 13-8. Typical Load File Action

Secondary Sort Processing

The flowchart for Secondary Sort Processing is shown in Figure 13-9, while the actual code is shown in Figure 13-10. Again, the processing is short as much of it is done in the SECSRT module, located starting at line 6000.

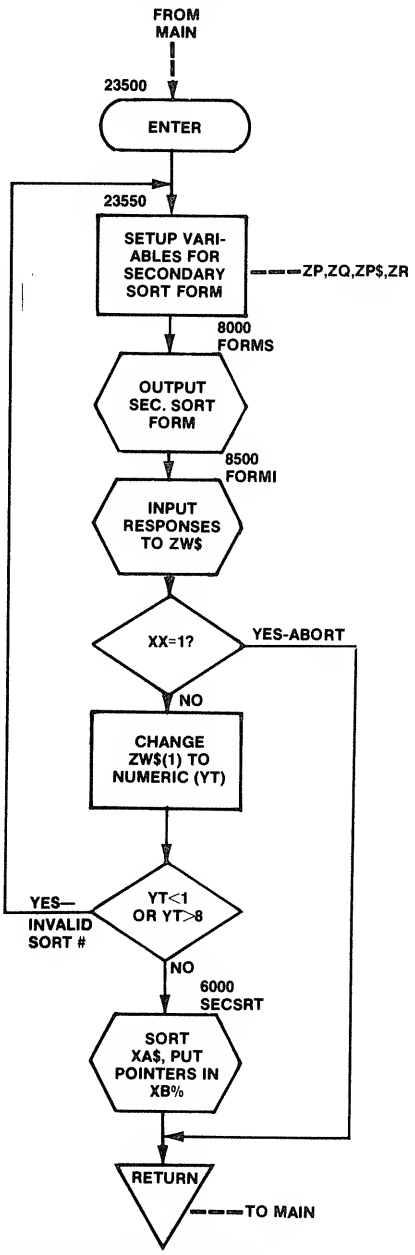


Figure 13-9. Secondary Sort Module Flowchart

```
23500 GOTO 23550 'MFSEC
23510 '*******************************************************
23520 ' THIS IS SECONDARY SORT PROCESSING
23530 '*******************************************************
23540 'FIRST OUTPUT SORT SKELETON
23550 ZP=55:ZQ=1:ZP$(0)="SECONDARY SORT":ZP$(1)="SORT ON FIELD # :"
23560 ZR(1)=1:GOSUB 8000
23570 GOSUB 8500:IF XX=1 GOTO 23620
23580 YT=VAL(ZW$(1))
23590 IF YT<1 OR YT>8 GOTO 23550
23600 'NOW SORT
23610 GOSUB 6000
23620 RETURN
```

Figure 13-10. Secondary Sort Module Listing

The Secondary Sort form is first displayed by a call to FORMS (line 8000). The field number for the sort is returned from FORMI (line 8500). YT is set equal to the VALue of ZW$(1) and then compared for a valid range of 1 through 8; the form is again output if the response is incorrect.

The last action of MFSEC is to call SECSRT to actually sort the XA$ array by adjusting the pointers in XB%. As previously discussed, this sort is rather time consuming compared to the entry-time sort of the primary field in XA%. The actions of SECSRT (line 6000) are described in detail in the previous section. On RETURN from SECSRT, the MAIN driver is reentered for the next menu selection.

Typical secondary sort processing is shown in Figure 13-11.

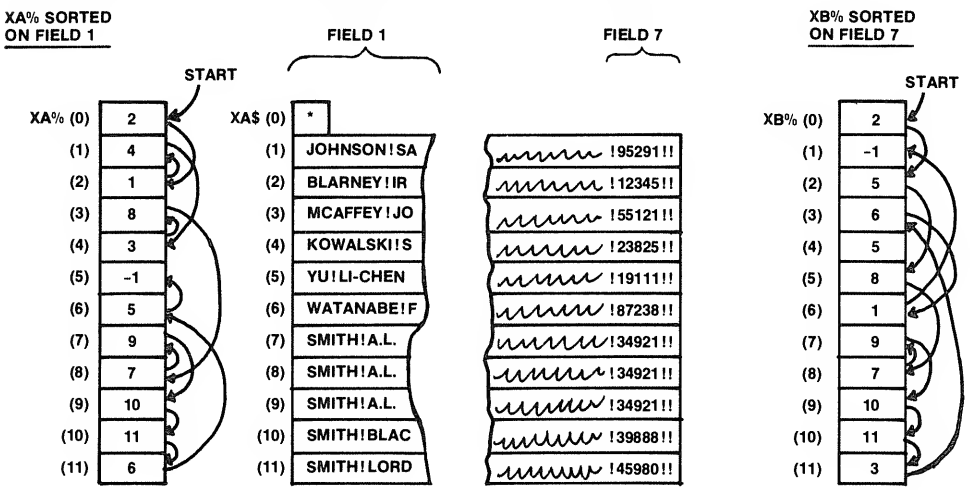


Figure 13-11. Typical Secondary Sort Processing

Search Processing

The flowchart for Search Processing is shown in Figure 13-12, while the actual code is shown in Figure 13-13. Search file is a "convenience" feature added to

the MAILLIST applications that allows a user to find any character string in a MAILLIST entry, as for example, a search for all entries that live in West Racoon, Wisconsin. All of the processing is done within the MFSRCH as there is no GPM module to perform this function.

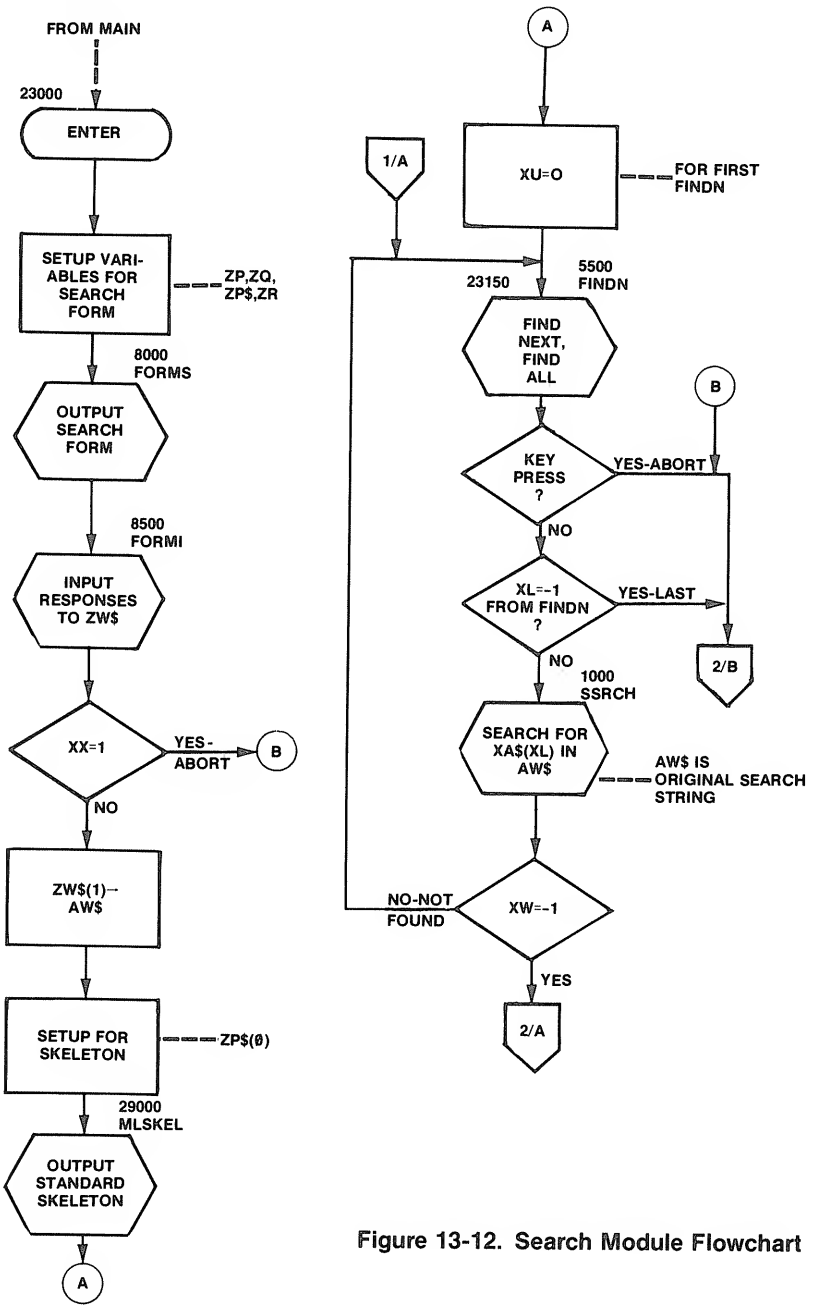


Figure 13-12. Search Module Flowchart

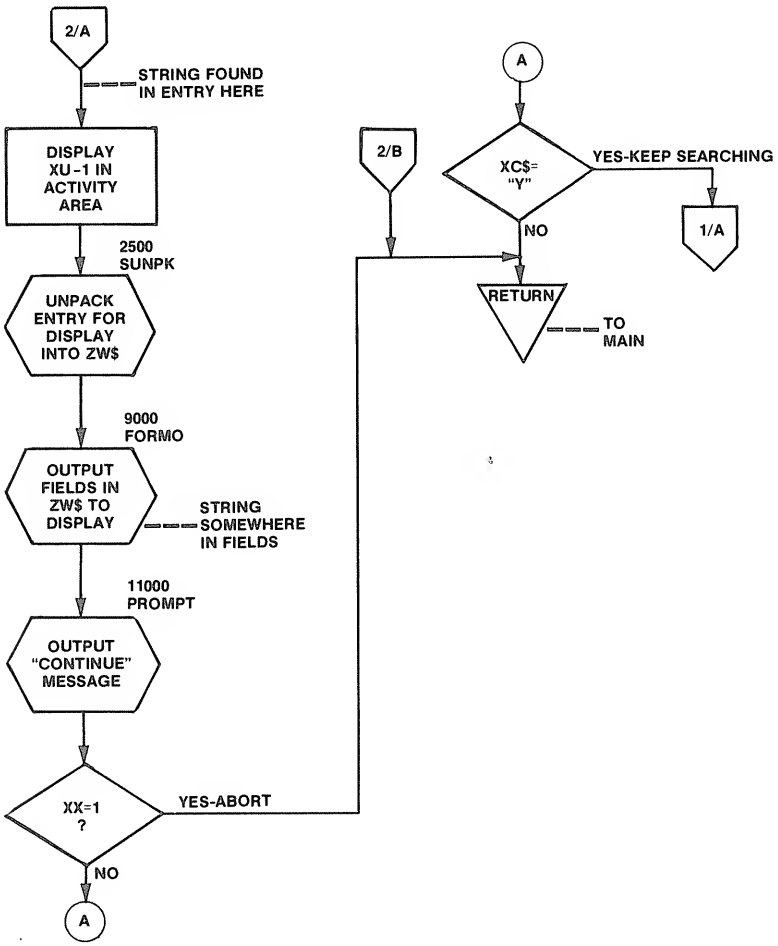


Figure 13-12. Search Module Flowchart (cont.)

First the Search form is displayed by a call to the FORMS module (line 8000). The Search form is shown in Figure 13-14. Next, the response to the form is read into ZW$(1) by a call to FORMI. If variable XX is set to 1 after the call the user has pressed CLEAR (Model I/III) or "up arrow" to terminate the Search operation and a RETURN is made at line 23270.

```
23000 GOTO 23050 'MFSRCH
23010 '*************************************************************
23020 ' THIS IS SEARCH PROCESSING
23030 '*************************************************************
23040 'FIRST OUTPUT SEARCH SKELETON
23050 ZP=55:ZQ=1:ZP$(0)="SEARCH":ZP$(1)="SEARCH STRING:"
23060 ZR(1)=30:GOSUB 8000
23070 'NOW READ IN SEARCH STRING
23080 GOSUB 8500:IF XX=1 GOTO 23270
23090 IF ZW$(1)="" GOTO 23270
23100 AW$=ZW$(1)
23110 ZP$(0)="SEARCH"
23120 GOSUB 29000
23130 'NOW READ IN ENTRIES ONE AT A TIME
23140 XU=0
23150    XS=-1:XT=1:GOSUB 5500
23160    IF INKEY$<>"" GOTO 23270
23170    IF XL=-1 GOTO 23270
23180    XW$=AW$
23190    XZ$=XA$(XL):GOSUB 1000
23200    IF XW=-1 GOTO 23150
23210    'STRING FOUND HERE
23220    PRINT @ YL, XU-1;"    ";
23230    XY$=XZ$:GOSUB 2500
23240    GOSUB 9000
23250    XB$="CONTINUE?":XB=2:GOSUB 11000:IF XX=1 GOTO 23270
23260    IF XC$="Y" THEN GOTO 23150
23270 RETURN
```

Figure 13-13. Search Module Listing

If ZW$(1), the "search string" is a null, and the Search processing is terminated by a GOTO 23270. Otherwise, the search string is transferred to "working variable" AW$ from ZW$(1). The MAILLIST skeleton is then displayed by a call to MLSKEL at line 29000. The skeleton only has to be displayed once, as FORMO will be called to display any entry containing the search string.

The loop from line 23150 through line 23260 is used to read in each entry of XA$, starting from the first in sequential order. The order, of course, is established by the pointers in XA%. Each entry is searched for the search string in AW$. If the search string is found, the entry is displayed on the form by a call to FORMO. The user then has the choice of continuing or terminating the search.

The first action in the loop is to call FINDN at line 5500 to find the next entry. XU is initially set to 0 (current number), XS is set to -1 (find all), and XT is set to 1 (find next) and the call to FINDN is made. On RETURN from FINDN, if XL=-1, then the last entry has been found and the search is over; a GOTO 23270 returns to the main menu in this case. The state of INKEY$ is also checked after the RETURN from FINDN. If INKEY$ is not "null" (""), the user has pressed a key, and the search operation is terminated by a GOTO 23270.

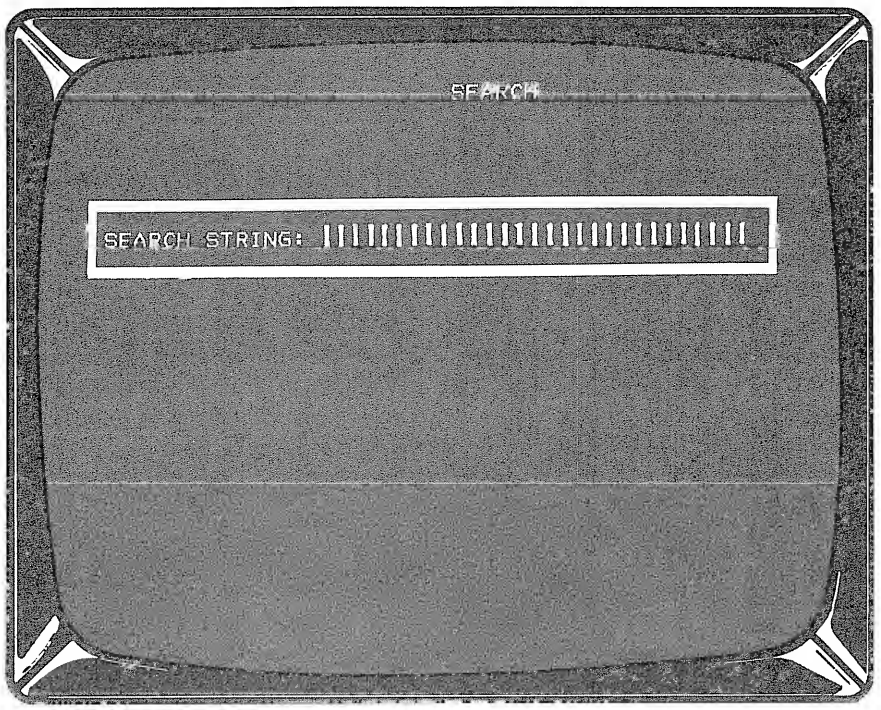


Figure 13-14. Search Processing Form

If the last entry has not been found and if the user has not terminated the search, XW$ is set equal to AW$. XW$ is the input variable to SSRCH, the string search module at line 1000. XZ$ is then set equal to the current entry XA$(XL). XL points to the current entry from the FINDN call. XZ$ is the "string to be searched" variable for the SSRCH call. The call to SSRCH is then made.

On RETURN from SSRCH, XW is set to -1 if the search string was not found, or to a positive character position if the string was found. If XW is -1, a loop back to line 23150 continues the search with the next entry. If XW is positive, the string was found in the current XA$ entry, XA$(XL). The number of the entry is displayed in the "activity area." The number is XU-1; XU is the "next entry" number from FINDN. XY$ is then set equal to XZ$, the current entry string, and the SUNPK module at line 2500 is called. SUNPK unpacks XY$ into ZW$(1) through ZW$(8). A call to FORMO at line 9000 then outputs the fields in the ZW$ array to the field positions on the screen.

After the fields have been displayed, the message CONTINUE? is displayed in the prompt message area by a call to PROMPT (line 11000). If the response is "Y," a loop back to line 23150 searches the next entry. If the response is "N," or if the

user has pressed the "CLEAR" key (Model I/III) or the "up arrow" (Model II), the Search processing is also terminated, and a RETURN is made to the main menu.

Typical search file action is shown in Figure 13-15.

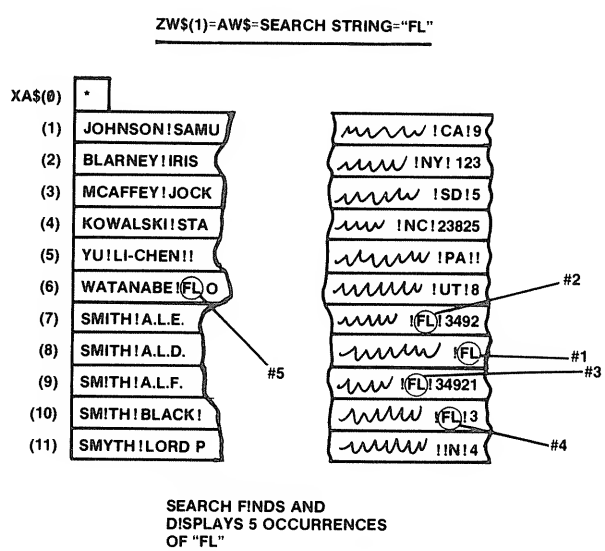


Figure 13-15. Typical Search Processing

Section IV
Other Business Applications

Chapter Fourteen
Information Retrieval
Over Multiple Disk Files

In this section we'll show you some other business applications programs built around the General Purpose Modules.

The first of these will be similar to the concept of the MAILLIST we've been working with. It will retrieve records by searching for "keywords," but it will do it over the entire disk, if desired. This program could be used for a variety of business applications including filing telephone conversations in a computer log, filing legal briefs for future reference, establishing a publications index, or others. The emphasis here will be on use of the entire disk "resource," or disk space.

The second application will be a simple inventory system that is geared to a file referenced by user part numbers. Records will include the part number, description, number on hand, and other "parameters," and simple "transaction records" will handle such things as receiving new shipments and orders.

The applications will be presented in the form of general design and flowchart descriptions, rather than complete BASIC programs. There are several reasons for this. The first is that each application would probably require one-half a book to present! The second is that defining a complete application is probably going to be useless to your specific needs. From this point on, it will be primarily up to you to design and implement your own business applications the way you would like to see them. If you can use the General Purpose Modules, or a portion of them, well and good. If not, we hope that the exercises in design and programming will be beneficial in implementing your own applications. (The third reason is that the author needs R&R after struggling through the GPM code . . .)

Problem Number 1: An Information Retrieval System

Suppose that we want to write an applications program using the General Purpose modules that will "retrieve information." Actually, almost every program does that, but we mean one in which a great volume of information may be stored with many "entries," with the capability to quickly retrieve all entries that contain references to a particular topic.

An example of this application might be a log of telephone conversations. While the conversation was going on the computer could be used to let the user enter a brief summation of the subject, the date, the caller, and so forth. This entry could then be filed in a DOS file. At a later time, all entries corresponding to "12-12-80" or "extortion attempt" could be retrieved and displayed or printed on a report.

Another use for a program of this type might be an index of publications. The title of the magazine would be entered, along with the date of issue, author name, title of article, subjects covered, and so forth. Later, all entries corresponding to a certain topic could be printed out or displayed. Typical entries for this use are shown in Figure 14-1.

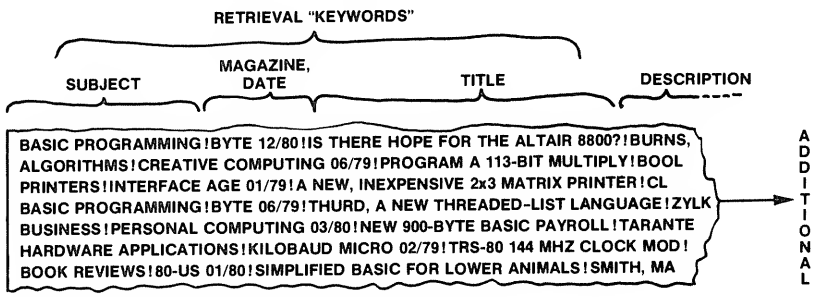


Figure 14-1. Typical Publications Index Entries

If we are to use the GPM modules for this application, we have the mechanism for displaying menus, outputting and inputting data on forms, building an ordered list in memory, adding to, deleting, or modifying entries on the list, saving the list on disk, and loading it back in from disk. The GPM is set up for ordering entries by field number 1, which could correspond to a user designated "keyword," as shown in Figure 14-2. Searches could be made on other keywords in similar fashion to the "Search" function of MAILLIST.

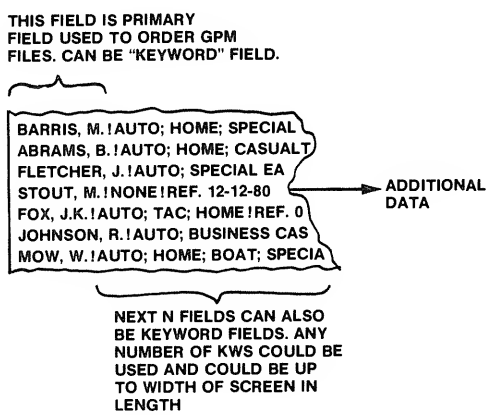


Figure 14-2. Keyword Ordering

One problem that is present, however, is the use of disk files. How can we expand the disk "resource" so that we do not have to limit the disk file to a size that can be contained in memory with the GPM modules and application program? After all, we would like to use all 90,000 bytes of disk on the Model I, all 180,000 bytes of disk on the Model III, and all 500,000 bytes on the Model II.

We might even want to use more than one drive, to expand the disk capacity to more than four times our capabilities in a small system.

Using the GPM Modules With Additional Disk Files

The CDLOAD and CDSAVE modules in the GPM are geared toward a single file. This file is loaded into memory from disk by a user-specified name. The file cannot be larger than the memory size left in RAM after the GPM, applications program, variable storage area, string storage area, and so forth are established.

The obvious answer here is to use more than one file — to use as many as are convenient on disk. This will serve several purposes. It will enable the GPM to work with smaller files in memory, avoiding problems of "pushing the limit" on string storage space. It will also expand the amount of disk space used to the limit of the disk by using more than a single file.

Of course, it's easy to see how more than one file could be used even with the existing structure. In MAILLIST, for example, we could have a separate file for A-Z — twenty six files in all, or we could separate the names into other convenient groupings. What we really want in an applications program, however, is a way to handle multiple files automatically, without manual intervention. What we'll show you in the following design and flowchart will define one approach to how this can be done.

KEYWORD Design Spec

Every program has to have a name, and we'll cleverly call ours KEYWORD, as information is retrieved based on three keywords. A portion of the design spec for KEYWORD is shown in Figure 14-3-1.

KEYWORD Design Specification

OVERVIEW:

KEYWORD is an information retrieval program that allows the user to store short records of information (up to 200 characters per entry) on disk file. Records may be easily added to or deleted from the disk. At any time, all entries that contain a specified keyword may be retrieved and displayed or printed.

LOADING PROCEDURE:

To run KEYWORD, first load BASIC. If you are using the Model II, specify BASIC -F:1 to allow you to have at least one disk buffer. Next, load KEYWORD from disk by RUN KEYWORD. KEYWORD should start execution, and you should see the display shown in Figure 14-3-1. Now enter the model number of your system, 1 for Model I, 2 for Model II, or 3 for Model III. KEYWORD will continue initialization procedures as shown by the activity display in the upper right-hand corner of the screen. There will be no display of the 1, 2, or 3 digit.

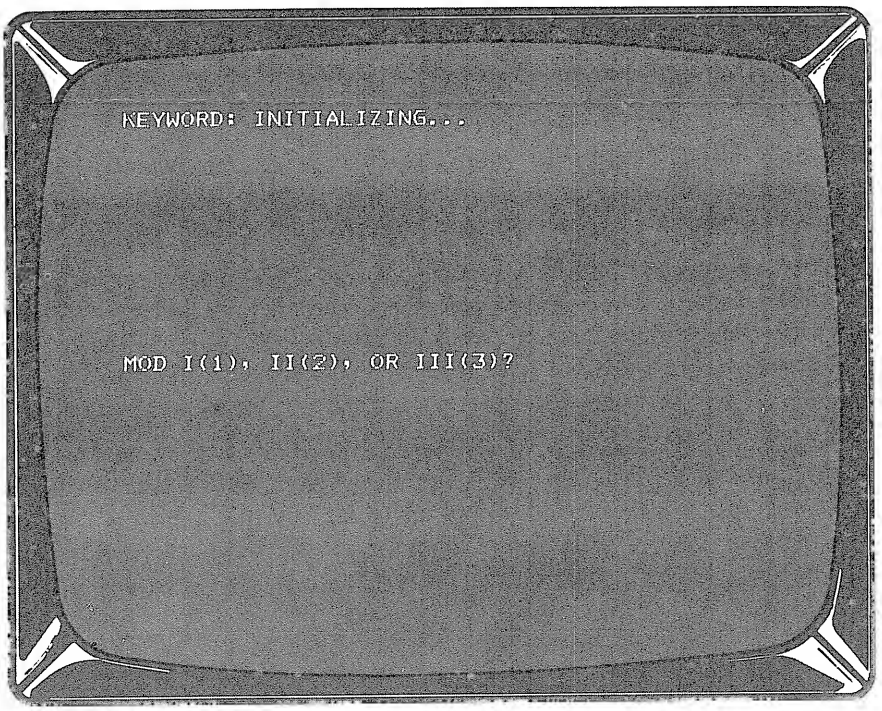


Figure 14-3-1. Loading KEYWORD

After initialization, KEYWORD will display a menu of items as shown in Figure 14-3-2.

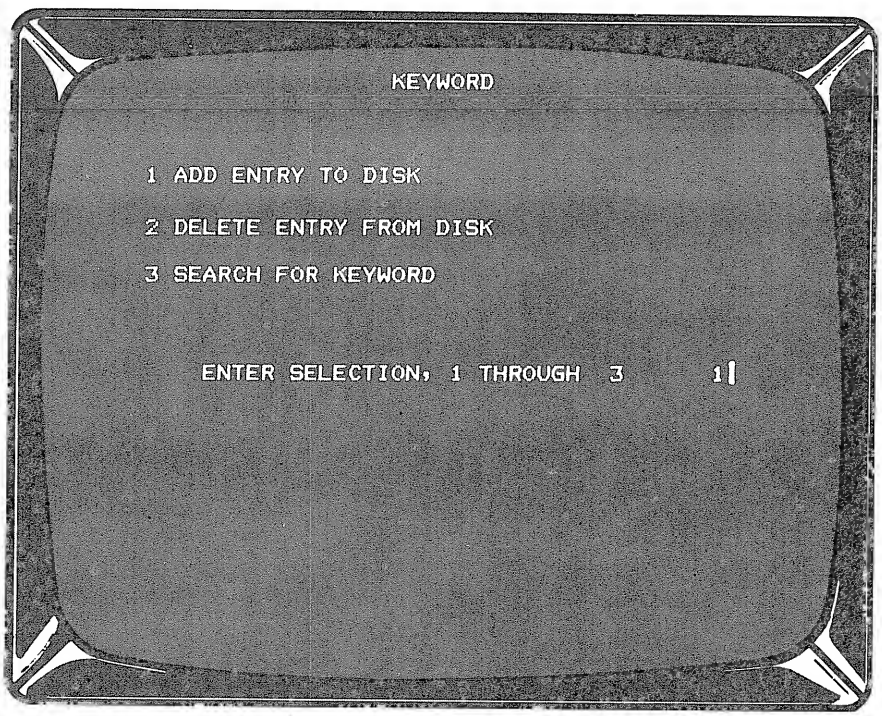


Figure 14-3-2. KEYWORD Menu

PROGRAM FUNCTIONS:

The separate program functions for KEYWORD are shown in the menu. They are:

Function 1: Adds an entry to disk.
Function 2: Deletes an entry from disk
Function 3: Searches for a keyword on disk.

Adding an Entry to Disk

To add an entry to disk, menu function 1 is selected. After selection, the form shown in Figure 14-3-3 is displayed. The bulk of the form is devoted to four fields (TXT:) of text of fifty characters each for a total of 200 characters of text. The text can be any appropriate text for the data being stored. If you are keeping a record of telephone conversations, for example, you might have a brief resume of the subject, date, party, and so forth. If you are keeping a record of publications, you might have the magazine, issue date, title of article, author, and so forth.

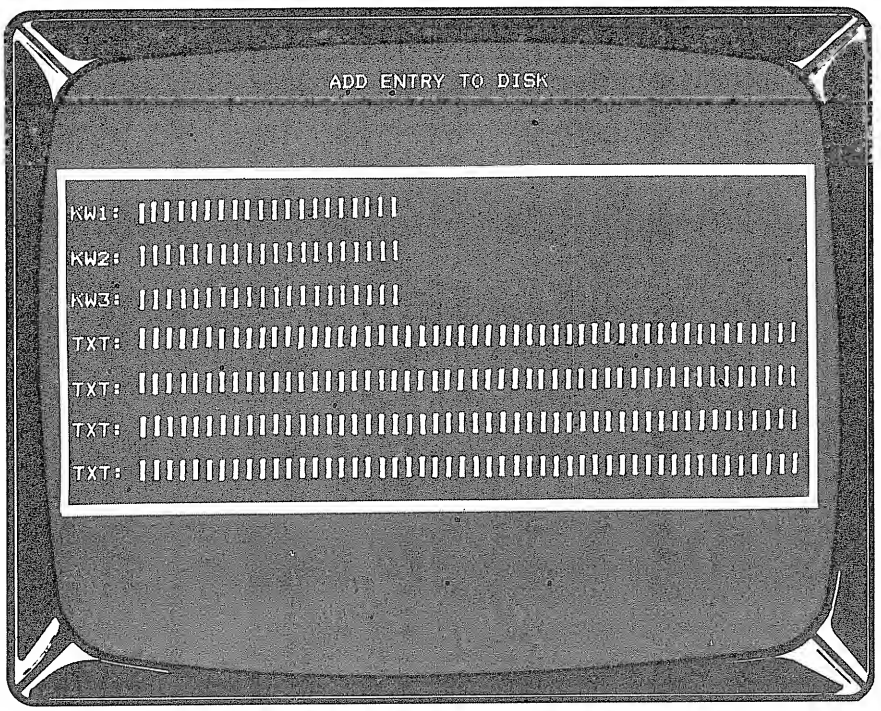


Figure 14-3-3. Add Entry Form

The first three fields are the "keywords" under which the data is to be filed. The first of these (KW1) is the most important. It is the main keyword for filing. **The first character of this keyword must be A-Z.** Except for that restriction, keywords may be any combination of alphabetic, numeric, or special characters including spaces. All "records" in the file will be ordered on this keyword, and retrieval of records containing this keyword will be very fast.

The two remaining keywords are secondary keywords that may also be used in retrieving data from disk. Retrieval times for records containing these keywords will be much longer than retrieval of records containing the main keyword, but they are included to make the application more useful.

To use the Add entry, enter all three keywords, with the primary keyword in field 1. If you want to use just one or two keywords, simply press ENTER for KW2 or KW3 or both. After entering the keywords, enter the text by typing in any characters except "!". If the text will fit on less than four, three, or two lines simply press ENTER for the other lines. After entering the text, the record just defined will be added to the disk file. A typical entry is shown in Figure 14-3-4.

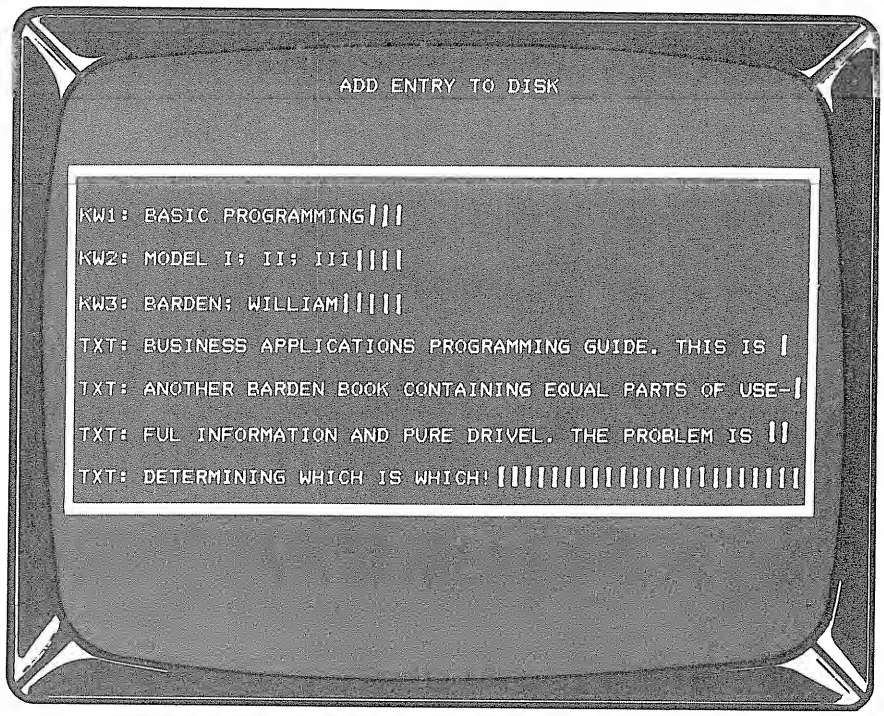


Figure 14-3-4. Typical Entry

Deleting an Entry

To delete an entry from disk, select menu item 2. The form shown in Figure 14-3-5 will be displayed. Enter the primary keyword that defines the record.

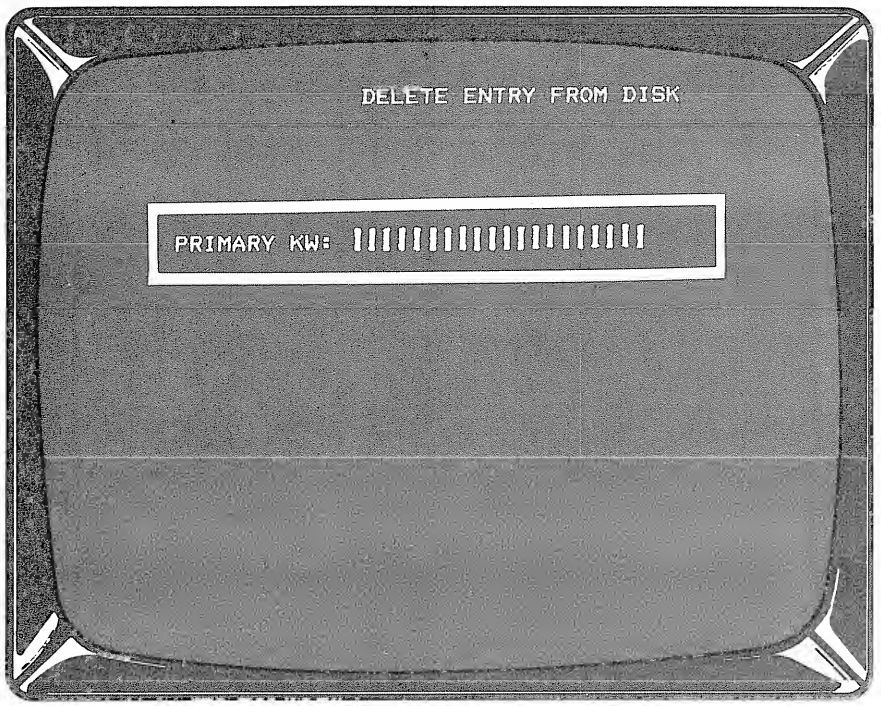


Figure 14-3-5. Delete Entry Form

The first record containing the keyword will then be displayed on the screen as shown in Figure 14-3-6, along with the message DELETE (D) OR NEXT (N)?. To delete the record, press D. The record will be deleted from the disk file, and the next record in sequence will be displayed.

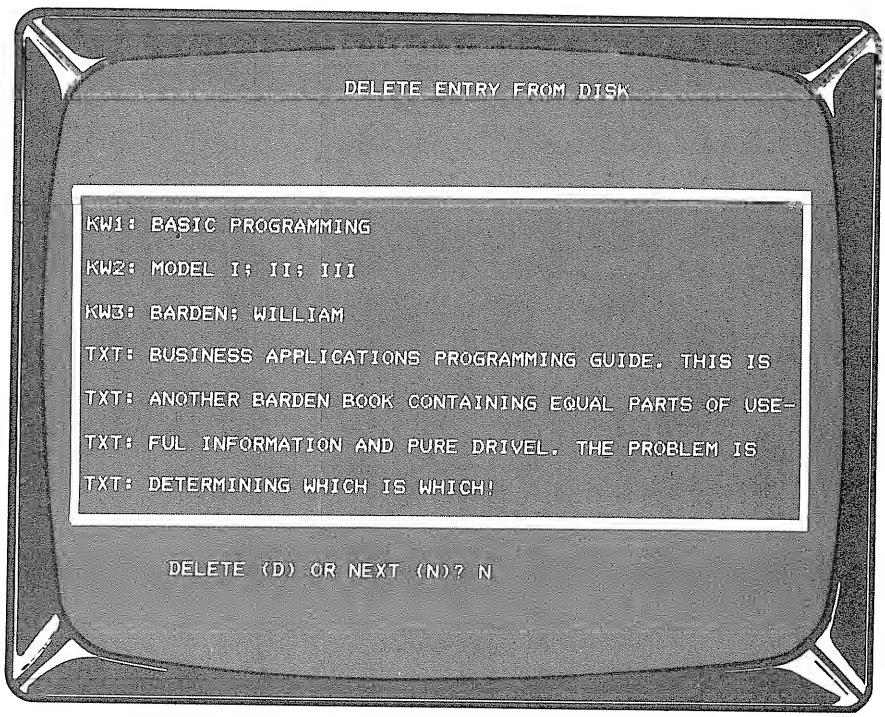


Figure 14-3-6. Delete Entry Example

To bypass deleting the current record, but to display the next record, press N. The next record in sequence will be displayed on the screen, and the process can be continued until the desired record is found.

To return to the menu, press CLEAR (Model I/III) or "up arrow" (Model II). The menu will be redisplayed for the next choice.

Modifying a Record

Since the records are short, modifications of records are handled by manual deletions of the erroneous record, and adding the corrected entry again in an "Add Entry" operation.

Searching for a Keyword

If menu item 3 is selected, KEYWORD will display the form shown in Figure 14-3-7. Enter the KW field. KW should be the "primary" keyword, the one under which the record was filed initially. This may not be possible. If telephone conversations are filed by date in the KW1 field, for example, and a search is to be made

for a name from the KW2 field, then one of the two secondary keywords must be entered as the "search" key.

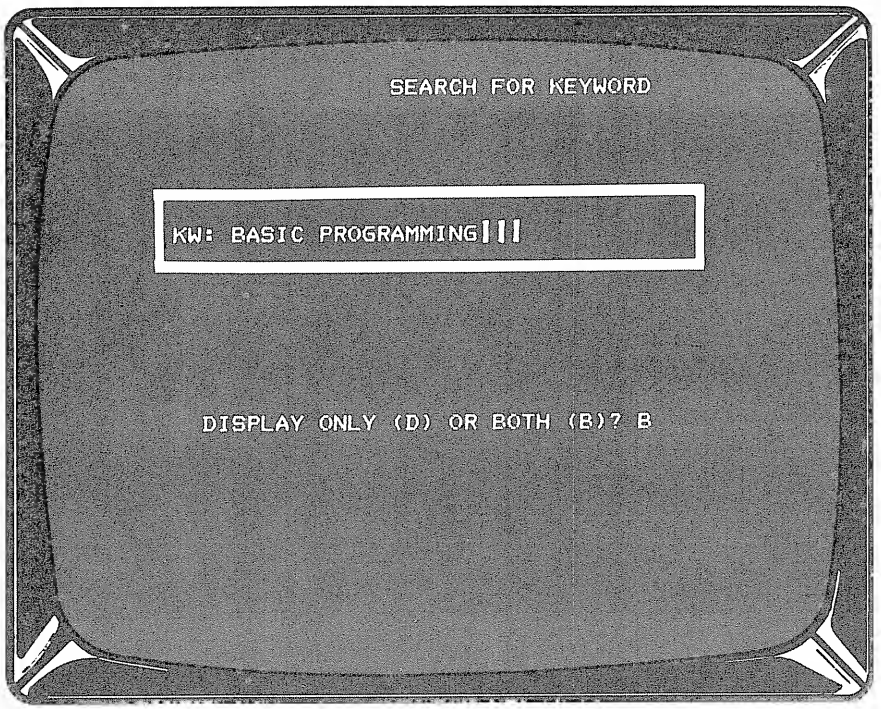


Figure 14-3-7. Search Form

Enter D for display only or B for both display and printing, depending upon the desired print action.

KEYWORD will now search the entire disk for records that have "primary" keywords corresponding to the search keyword. It will display and/or print all such entries as shown in Figure 14-3-8. After each entry has been displayed, the message NEXT (N)? will be displayed. To find the next record, press N. To return to the menu, press ENTER (Model I/III) or "up arrow" (Model II).

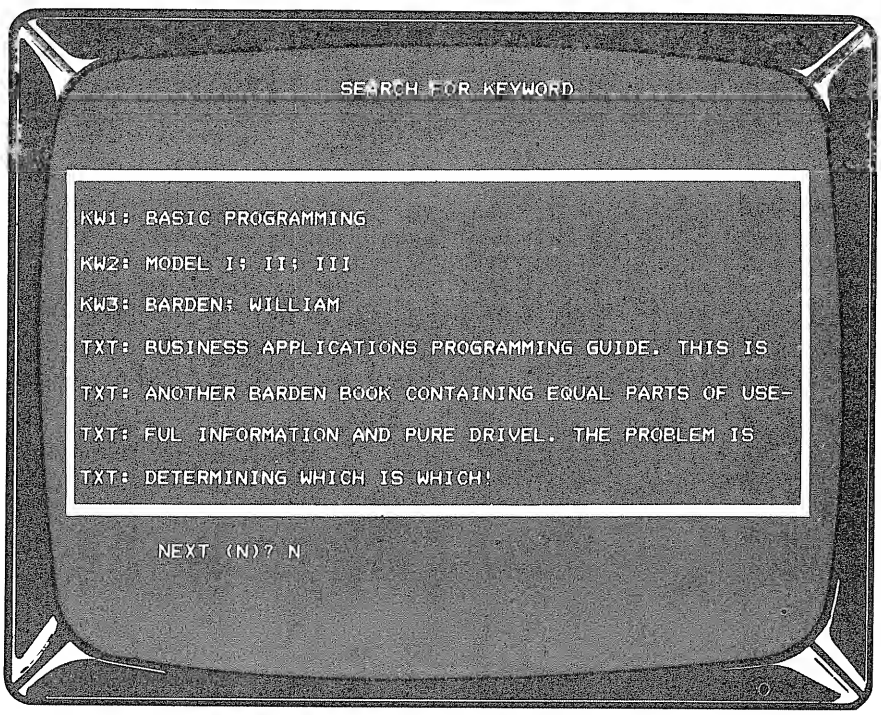


Figure 14-3-8. Search Display

After KEYWORD has searched and displayed all records containing the search key in KW1, it will search for the key in all KW2 and KW3 fields. The same display will be used as previously.

Pressing any key during search **processing** (non-display) will cause the search to terminate.

General KEYWORD Design

We generated the design spec above as an example of "top-down" design for a BASIC applications program. This is the "first cut." It's quite possible that we may want to add additional functions or change the operations above once we start flowcharting KEYWORD. Implicit in the production of the design spec were the steps of

- Learning the system
- Learning the BASIC
- Research into the application

Another consideration, of course, was incorporation of the existing General Purpose Modules. Also, during the design spec definition, we kept thinking ahead to the implementation about "easy" and "hard" things to do, and this influenced the general design. When it came time to incorporate a "modify" capability, we opted to modify by deleting and adding an entry manually, as the modify would not be (very) simple because of the file structure in use.

The key to the design of KEYWORD is the disk file structure. Twenty-six files are used, named "KWA" through "KWZ." Any time a primary keyword is referenced, the first character from the keyword (and it must be a letter) is added to the letters "KW" to make up the file name "KWx."

The maximum number of files on a Model I system is 64, so there is no problem with the **number** of files used, unless the diskette is used with a number of other files. As far as the actual storage, each file is a sequential file and new data in the file fills up remaining space; there are no "gaps" of data as there might be in a random file. The minimum space allocated by DOS for each file is one granule, or five sectors (Model I). The total "best case" minimum space is 5 times 26 sectors, or 130 sectors, or 13 tracks out of 35. Actually this is not really the minimum since a file is not used if a keyword containing the letter has never been referenced. If a primary keyword of "Zxxx" has never been used, for example, disk file "KWZ" is not allocated.

Assume that typical entries are 128 characters long. This would represent, say, 25 characters of keywords plus 7 "delimiters" (!) plus 96 characters of text. In this case, about two entries could be made per sector, allowing for a maximum of 460 entries in a TRSDOS diskette, the worst case (Model I). For a data diskette, the number of entries would be increased to about 648 (Model I). For a Model III system the maximum number of entries would be close to 1300, while a Model II could hold 3900 entries.

Of course, there are problems in allocating disk files based upon the first letter of the keyword. Certain letters are used more frequently, and these files would have more entries. However, assume that the average file would have 20 entries while the largest file has 40 entries. The average file would take about three seconds to read in, another three seconds to add the entry, and three seconds to write out. These times are within acceptable limits.

In the following flowcharts, we haven't "optimized" the file manage. The files could have been split up on different disk drives by using a disk drive specification in the file name, such as "KWS:1." This would permit larger overall files (at increased processing times) or even expanding the number of files to forty or fifty (at decreased processing times). Perhaps selection of the file could be based on a "table" of names — keywords with first characters of X-Z would use file "KWX," keywords with first characters of I, J, or K would use file "KWI," and so forth.

Another obvious improvement would be to never read in a file if the current file in memory is the same file, and to write out an updated file only if a new file was to be read in. This would be handy in cases where many keywords with the same first character were to be processsed.

The point here is that there are many games to play with disk file allocation; the approach in KEYWORD is one possible solution.

KEYWORD Flowchart

The flow chart for KEYWORD is shown in Figure 14-4. It is divided into five sections, the MAIN driver, Add Entry, Delete Entry, Search, and the DISPRT subroutine.

MAIN Driver

The MAIN driver first initializes all arrays and variables and then inputs the code for the type of system. The parameters for the system in use are then stored. The AINIT module is used here. Next, MENU is called with appropriate title and menu text in the ZA$ array. After MENU is called, a GOSUB to the appropriate processing routine may be made by testing variable ZB.

Add Entry Processing

The Add Entry Processing function reads in the proper file, adds the current entry, and writes out the file again. We don't care here if the file exists or not. If the file doesn't exist, an OPEN call in CDSAVE will create it. If the file exists, the OPEN call in CDSAVE will overwrite the existing file. DOS handles all of this automatically.

First of all, FORMS is called to display the Add Entry form (Figure 14-3-3). Next, FORMI is called to input the seven fields associated with the form into ZW$(1) through ZW$(7). If variable XX is a 1 after the FORMI call, the user has terminated the operation by pressing ENTER (Model I/III) or "up arrow" (Model II); in this case a return is made to the menu.

Next, the fields are packed into a single string by SPACK. The XY$ string will have "!" "delimiters" as in the MAILLIST case.

A check is made of the first character of KW1. If it is not A-Z, an error message such as INVALID KW1 NAME can be displayed by PROMPT, and the input process repeated.

If KW1 is valid, the ZZ array is setup for a "file not found" error code. The message associated with this error code may be null (""). This setup is necessary because we don't know at this point whether a file exists for the keyword. CDLOAD is called for "KWx." On RETURN, XX will be set if the file was not found or 0 if the file was found. We do not care, as the XA$ array is initialized in the first case (empty).

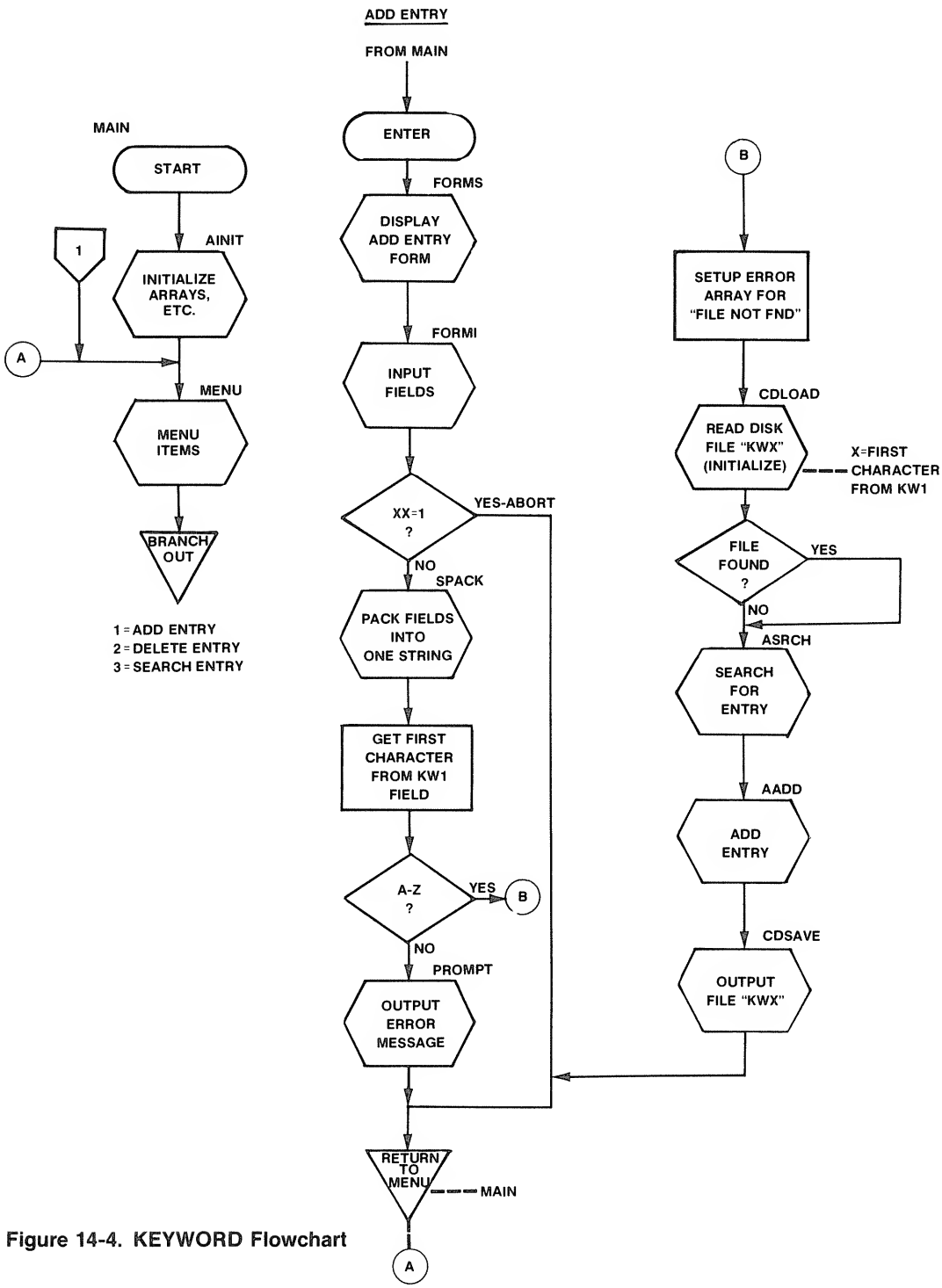


Figure 14-4. KEYWORD Flowchart

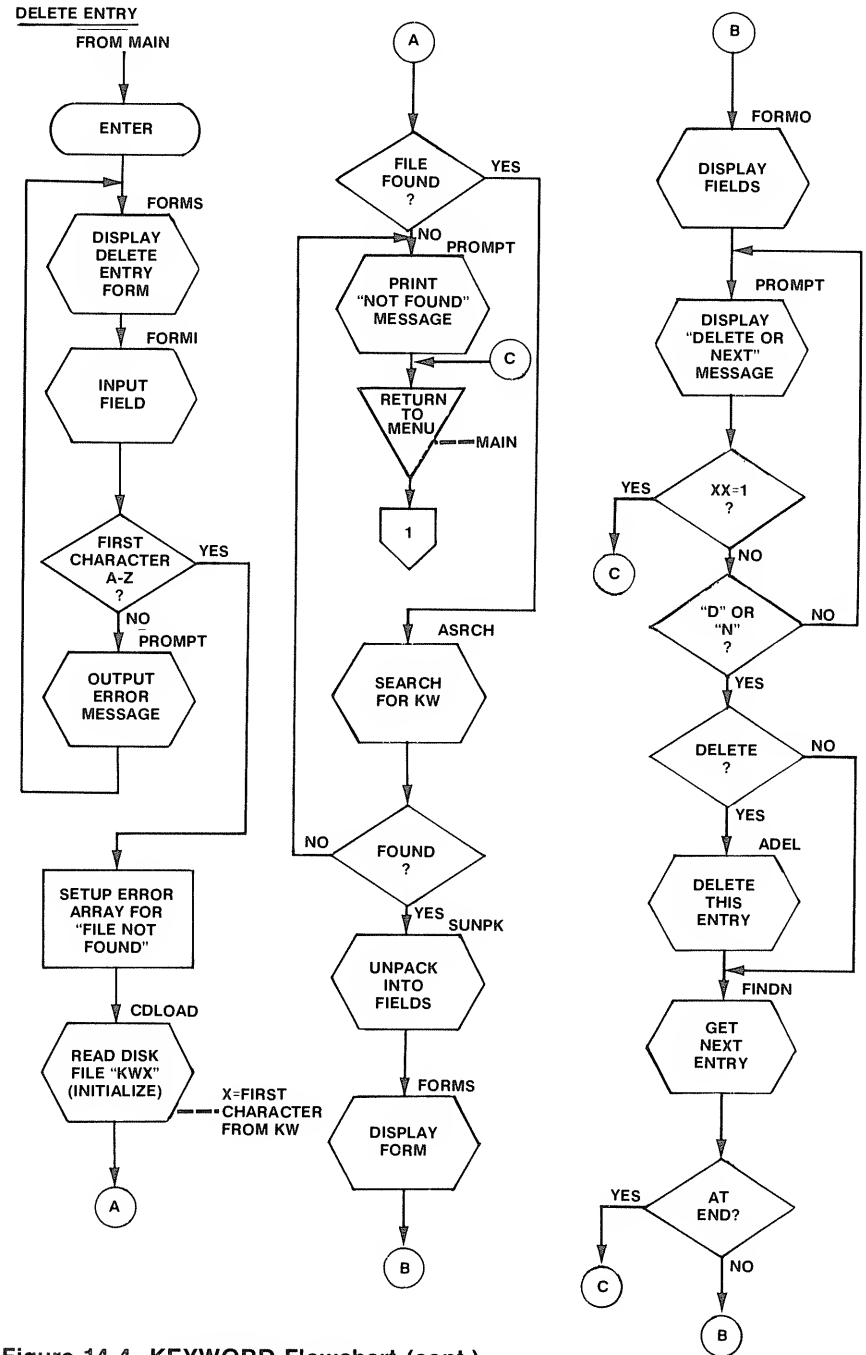


Figure 14-4. KEYWORD Flowchart (cont.)

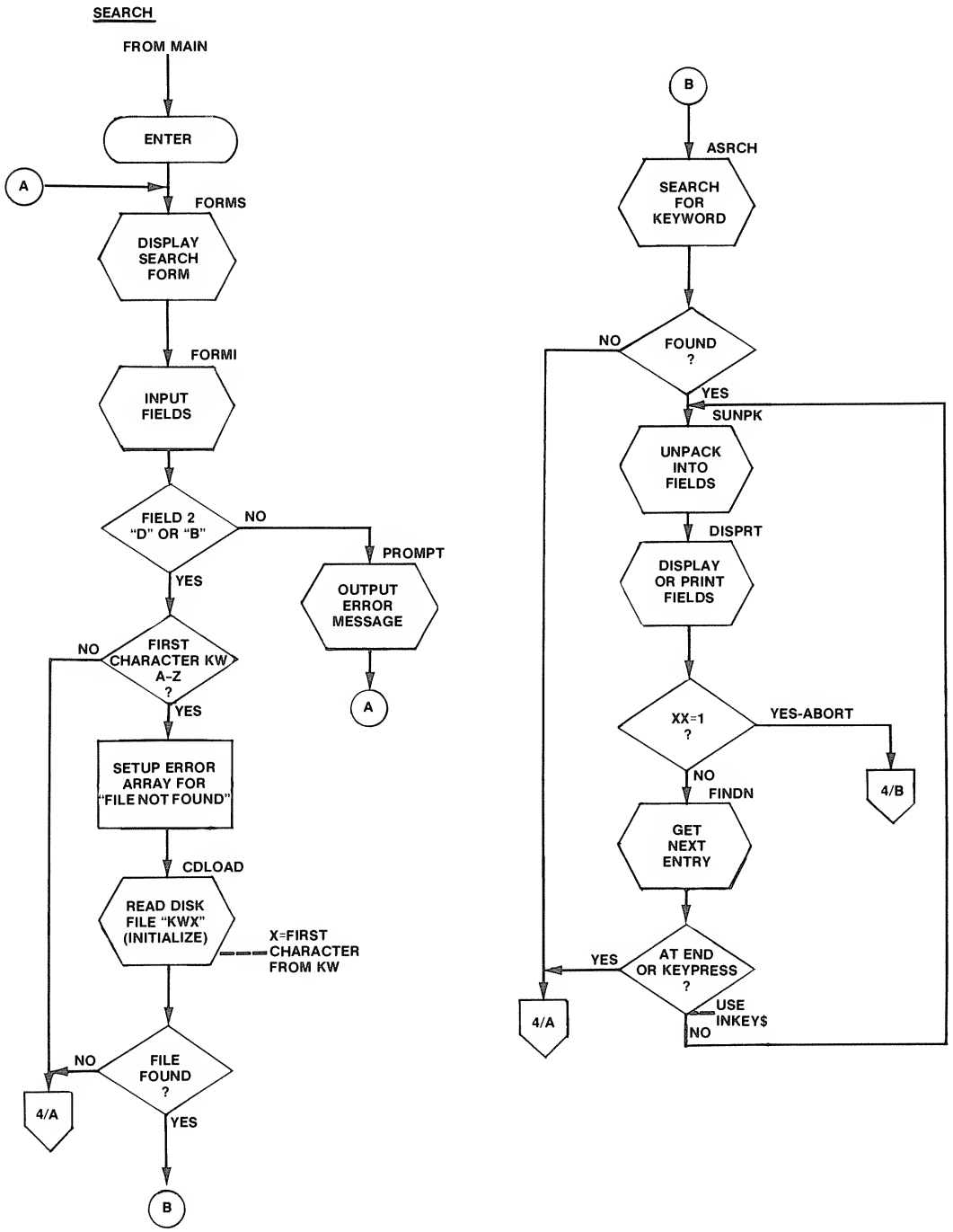


Figure 14-4. KEYWORD Flowchart (cont.)

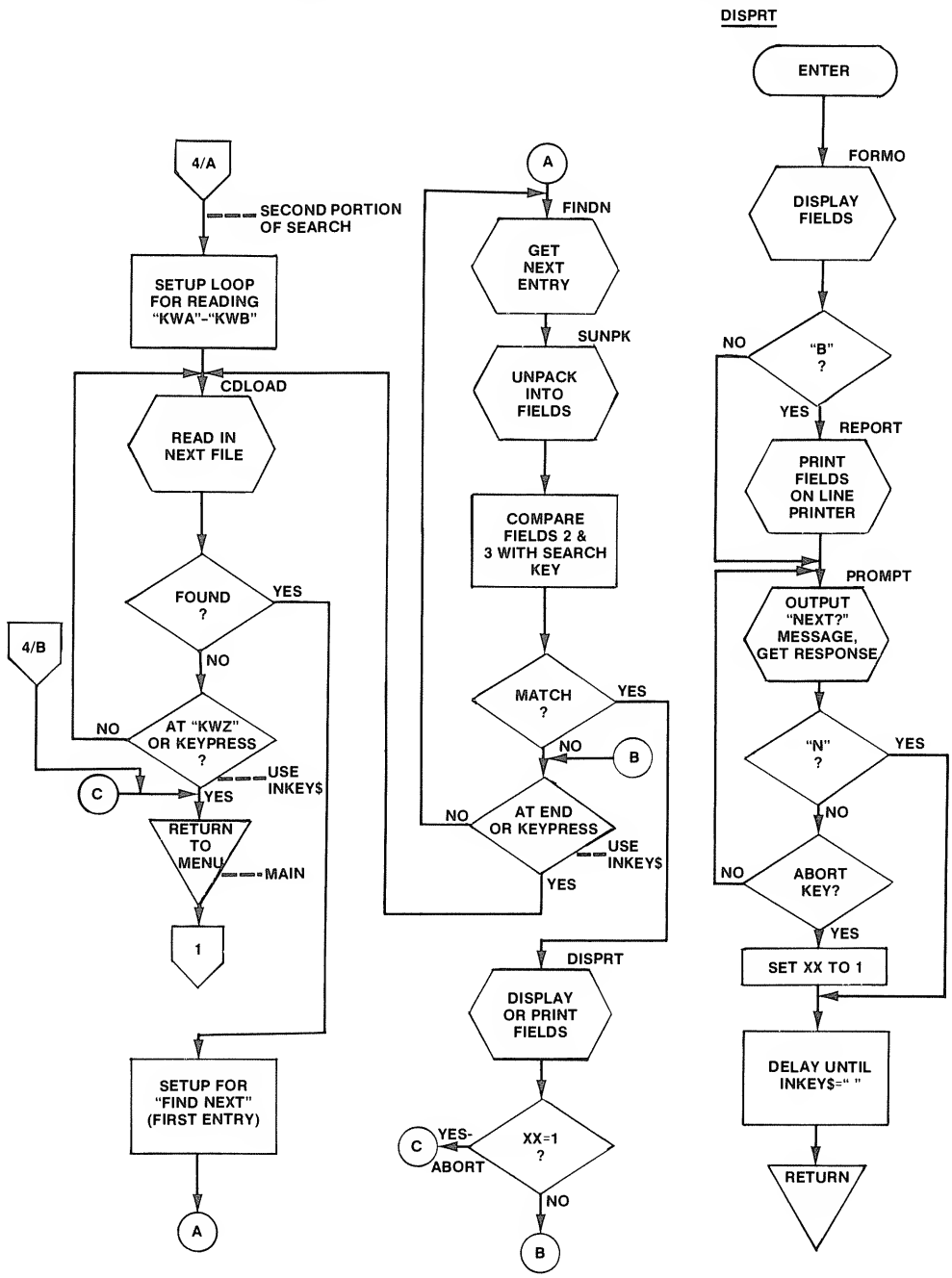


Figure 14-4. KEYWORD Flowchart (cont.)

Next, a search is made for KW1. The search variables XJ, XK, and XL (and others) are RETURNed with the proper values for adding the entry, even for an empty XA$ file. AADD is called to add the entry.

The last action taken is to write the disk file "KWx" back out again and to return to the menu.

Delete Entry Processing

Delete entry processing is more involved. Here the proper file is located based on the initial character of KW1. Then a search is made for the first entry containing the keyword. If the entry is found, the user may delete or go to the next entry. If he looks at more than one entry, the FINDN module is called to get the next entry. Delete Entry is geared to easily deleting one or more entries with the same primary keyword.

First of all, FORMS is called to display the Delete Entry form (see Figure 14-3-5). The field associated with the form is read in by FORMI. If the first character of the KW is not A-Z, an error message is displayed by PROMPT and the form output repeated.

If the first character of the KW is valid, the error setup for "file not found" is done similarly to the Add Entry Processing. CDLOAD is then called to load the file. If variable XX is set on RETURN, the file was not found, the entry cannot be deleted, and a NOT FOUND message is displayed by PROMPT, followed by a RETURN to the menu.

If the file was found, it is in XA$. We can now search for all entries containing the keyword. A search is made by calling ASRCH with XD$ containing the keyword. If an entry is not found (XL equal to -1), a return is made to the menu. Bear in mind that the ASRCH will search for the KW among entries that contain not only the KW, but additional keywords and text as well — an exact match will never be made, but XL will be returned to the first entry containing the KW string.

If the entry is found, it is unpacked into separate fields (ZW$) by SUNPK. Next, the fields are displayed on the form by FORMS (form output) and FORMO (field output). The message DELETE (D) OR NEXT (N)? is then displayed by PROMPT. A check is made for the correct response.

If the response is D, the entry is deleted by a call to ADEL. If the response is N, the entry is not deleted. RETURN can be made to the menu by a press of CLEAR (Model I or III) or "up arrow" (Model II).

A call is now made to FINDN to find the next entry from this point. If XL=-1 after this call, the last entry in XA$ has been reached and a RETURN is made to the menu. If XL is not equal to -1, the next entry is displayed, and the process repeated. This action continues until a "CLEAR" or until the last entry has been processed.

Search Processing

Search processing is split up into two parts. Searching may be stopped at any time by pressing any key during the search or by pressing CLEAR (Model I/III) or "up arrow" (Model II) during display of items. The first part is a search of the file associated with the search keyword. This involves reading in the "KWx" and searching for any and all entries whose first field matches the search key. The second portion systematically searches all files, KWA through KWZ, and looks for the search key in KW2 and KW3 of each entry in each file.

The FORMS module is first called to output the Search form (see Figure 14-3-7). The fields are input by FORMI. If the D or B response is not correct after FORMI, PROMPT is called with an error message, and the process repeated.

Next, the search KW is tested for a first character of A-Z. If a character is not found, the assumption is made that one of the other two keywords is involved, and a GOTO the second part of Search takes place. If A-Z is found, the error code setup is done as before, and CDLOAD loads in the appropriate file. If the file is not found, a GOTO the second part of the Search takes place as there is no primary keyword that will match. If the file is found the search continues.

ASRCH is called to search for the primary keyword. If the KW is not found, a GOTO the second part of Search is made. If the keyword is found in the first field of the entry, the entry is unpacked into fields and displayed on a form (see Figure 14-3-8) or printed by a call to the DISPRT subroutine.

If the next entry is to be found, FINDN is called to get the next entry. If XL=-1, the second part of Search processing is entered. If XL is not equal to -1, the next entry is displayed or printed. A keypress is also detected at this time. A keypress will abort the first search and cause the second portion of the search to be entered.

At this point, the first portion of the Search has been done. All entries in the disk file associated with the primary keyword have been displayed, if any. Now the Search will look for all entries in every file that contain the keyword in KW2 or KW3.

A loop is first setup for reading KWA through KWZ. This can easily be done by performing a FOR I=1 TO 26 statement and then forming the file name by ZP$(2)="KW"+CHR$(64+I).

Each of the 26 times through the loop, the next KWx file is read in. After KWZ (or a keypress), a RETURN is made to the menu. If the file is not found, the next file is loaded. If a file is found, the Search continues for keywords.

When a file is found, a setup is made for reading all of the entries in the file, starting with the first, by calls to FINDN, as in MAILLIST. FINDN is called to read in the next entry. This entry is unpacked into fields by SUNPK. The second and

third fields (in ZW$(2) and ZW$(3)) are compared with the search key. If either matches, the entry is displayed or printed by a call to DISPRT. If neither match, the next entry is found (unless XL=-1 or a keypress) in which case the next KWx file is read.

DISPRT Subroutine

The DISPRT subroutine displays and prints the entry fields. Either a D or B response to the previous display/print question causes a display by a call to FORMD. If the response is B, a call to REPORT also prints the entry. (The REPORT format must be properly setup by a definition of the items in the XP array. This should be done at the start of MAIN.) After the display or display and print, the message NEXT? is displayed by a call to PROMPT. Pressing an N key returns to the search processing to find the next entry; pressing CLEAR (Model I/III) or "up arrow" (Model II) aborts the search.

Chapter Fifteen
A Simple Inventory System

In this chapter we'll describe a simple inventory system that uses the General Purpose Modules as a base. The inventory program, INVENT, is not meant to be a sophisticated inventory system, but is designed to further expand the basic concepts of disk files and use of the GPM. A system similar to the one described **could** be used to advantage in a small business, however. The system described here permits:

- Establishment of a master inventory file with part numbers, description, number on hand, number on order, vendor, and other critical data
- Disk record keeping of the day-to-day transactions for the inventory — sales, incoming shipments, and orders
- Daily or periodic updating of the inventory file by generation of a new master file from the old master file and transactions
- Report generation describing the current contents of the inventory

Among the things we'll be discussing here is a very important point — how we can use GPM files, essentially strings of **character** data, to handle simple arithmetic computations. We'll see how it can be done easily with no strings attached . . .

Problem Number 2: An Inventory System

Suppose that we have a small business that has an inventory of several hundred or more parts. We would like to keep records of the current inventory on a day to day basis. Currently, by manual methods, we have inventory descriptions based upon part number.

Typical part information is as shown in Figure 15-1. The internal part number is a number from 1000 through 9999. The vendor part number is the part number that we would order from a vendor. The part description is a brief text description of the part. The list price is the price we would normally pay for the part.

PART #	VENDOR PART #	DESCRIPTION	LIST PRICE	NUMBER ON HAND	DESIRED LEVEL	NUMBER ON ORDER	DATE OF ORDER	VENDOR CODE
1234	74155	DECODER	1.25	9	20	0		NS
1235	74160A	COUNTER	1.15	5	20	10	12/12/80	NS
1238	74165	SHIFT REG	1.90	2	25	0		NS
1240	7485	COMPARATOR	1.60	7	10	10	1/2/81	TI

Figure 15-1. Part Information Example

The number on hand is the actual number in inventory at the current time. The desired stock level is the optimum level we want to maintain. If the actual level is higher than this, we are overstocked — lower, and we are understocked.

The number on order is the current number we have ordered but not received. The date of order is the date we placed the order.

The vendor code is a two-character code that references another list of vendors that contains the vendor name, address, contact, and so forth.

Each day we have **transactions** that directly concern our inventory. We may **sell** a number of items. Each time we sell items, we should adjust the inventory by subtracting the number of items sold from the current inventory amount. Each time we **receive** a shipment of parts, we should stock the part and adjust the inventory count upwards by the number of items received. We may also **order** parts if we see that our inventory stock is running low; we should make a note for the inventory item that we have ordered the part and how many we have ordered.

As we operate now, we have to visually inspect the inventory by riffling through cards that represent the current stock of parts. We have to make a judgment for each part concerning its status and take some action such as reordering.

How can we computerize this simple inventory system and improve upon the manual methods that we now use?

It seems apparent that we have all the tools necessary in the GPM to handle menus, form output, field input, sorting, adding and deleting entries, and loading and saving disk files. We've seen how that can be done in MAILLIST. The key here seems to be the day-to-day transactions — how can we "merge" the transactions with the old inventory data to produce an updated inventory file? Also, how can we perform the adds and subtracts on the inventory counts? Let's consider each question.

First of all let's establish a "master inventory file." This will be similar in concept to the other files we've been using. It will be made up of hundreds of records, or entries. Each entry will have a number of fields as shown in Figure 15-2. The primary field, field number 1, will be our internal part number. The other eight fields will be vendor part number, part descrition, list price, # on hand, desired stock level, # on order, date of order, and vendor code.

All of the information for a particular part will be character data, even the numeric quantities. All of the parts will be in a file called OLDMAST, for "Old Master." This file will be a sequential ASCII file identical to the structure we used in MAILLIST. Each entry of the file will be ordered on the first field, internal part number. The length of each entry will be variable, based upon the total number of characters describing each part. A portion of a typical OLDMAST file will look like Figure 15-3.

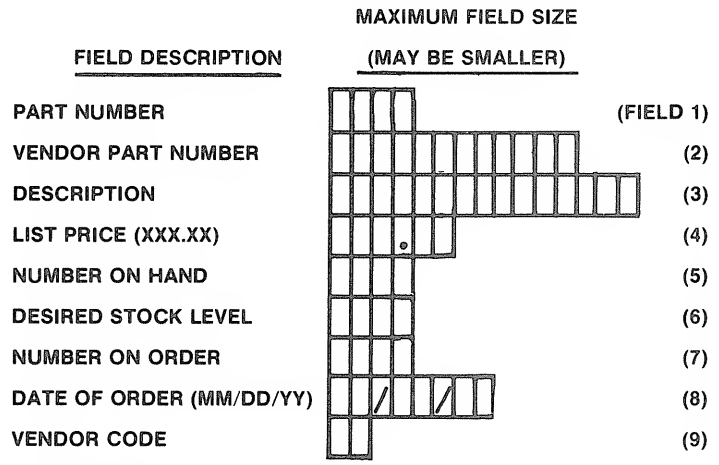


Figure 15-2. Master Inventory Records

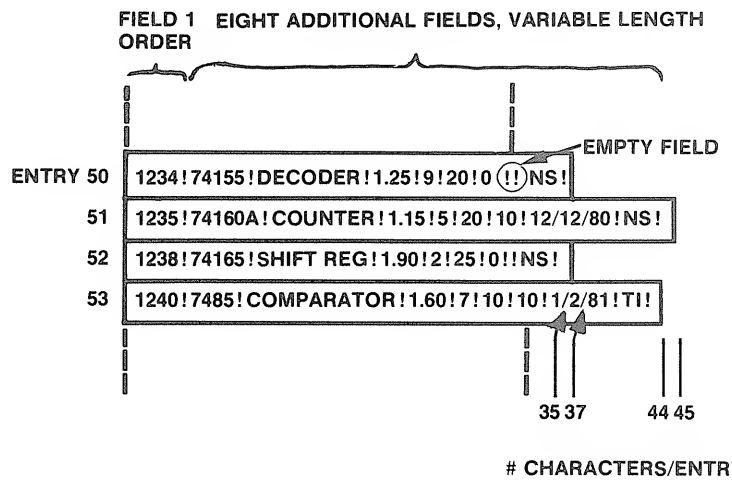


Figure 15-3. Typical OLDMAST File

In the INVENT program we'll have provisions for creating the OLDMAST file, just as we did in MAILLIST. Obviously we must be able to add part numbers to the file, delete them when we no longer carry them, and modify their descriptions when prices or vendors change.

We'll also establish a new type of file, called TRANS. This file will have the same format as GPM files — it will be a sequential ASCII file with a variable number of entries and with entries a variable length. Like the OLDMAST file, the first field of each entry in the file will be the internal part number. The other fields, however, will represent "transaction" data. We'll have three different types of

entries in the TRANS file — Sales, Orders, and Received. The fields for each of the three will appear as shown in Figure 15-4.

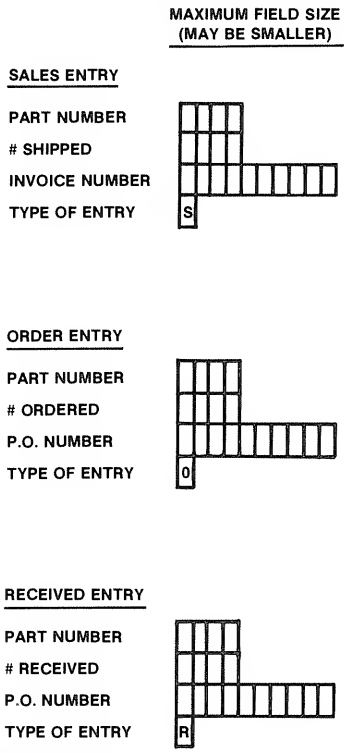


Figure 15-4. TRANS File Fields

The Sales entry will have the internal part number in field 1. Field 2 will be the number shipped or sold. Field 3 will be a reference to our invoice number. Field 4 will be an "S" for a sale transaction.

The Order entry will have the internal part number in field 1. Field 2 will be the number ordered. Field 3 will be a our purchase order number. Field 4 will be an "O" for an order transaction.

The Received entry will have the internal part number in field 1. Field number 2 will be the number received. Field 3 will be our purchase order number. Field 4 will be an "R" for a received transaction.

As parts are sold, ordered, or received, we will make entries into the transaction file. As the TRANS file will be ordered by part number, the three types of transactions will be intermixed, as shown in Figure 15-5.

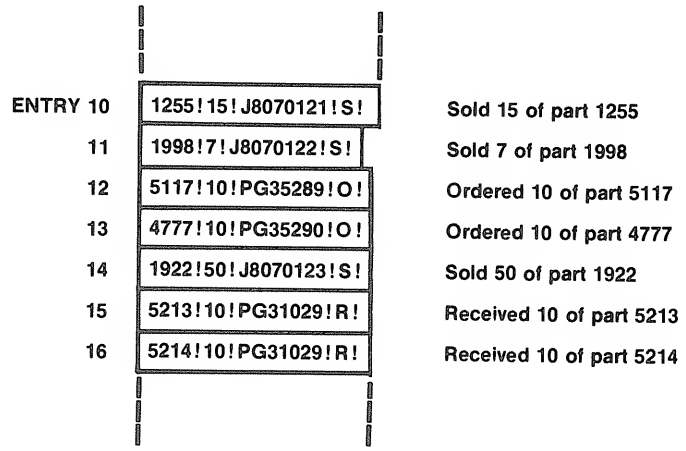


Figure 15-5. Typical TRANS File Entries

The daily activity, then, will be recorded in the TRANS file. At the end of the day, or other period, the transactions in the TRANS file will be merged with the OLDMAST.

The way that this will be accomplished is by first loading the OLDMAST by the CDLOAD module, and then by merging (M) the entries in the TRANS file. When this is done, we'll have a conglomeration of OLDMAST entries with TRANS entries. All entries, however, will be ordered by part number. All TRANS entries pertaining to the part number will be immediately adjacent to the OLDMAST entry. This situation is shown in Figure 15-6.

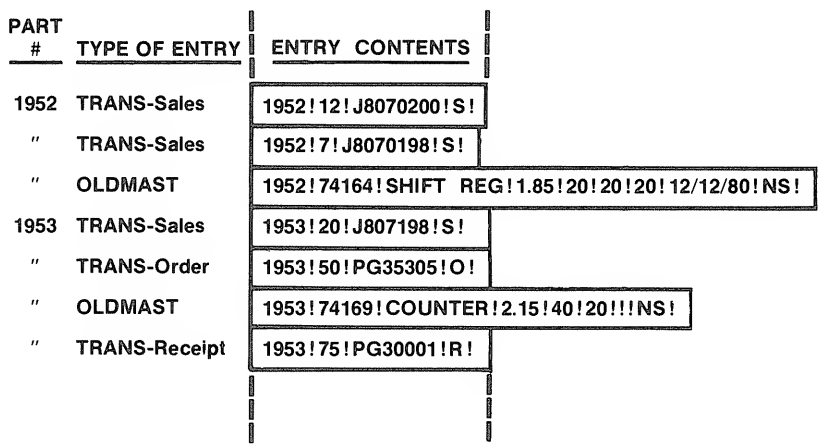


Figure 15-6. OLDMAST and TRANS Merge Operation

Now the file in memory can be processed by another part of the INVENT program. This part would take all TRANS entries for every OLDMAST entry and adjust the # on hand field, the number on order field, and date of order field based on the type of TRANS entry. As each TRANS entry is processed it can be deleted from the file. When every TRANS entry has been used for an update, the remaining OLDMAST entry will have the current information about the part. When the update is done for every entry, the field in memory will represent an updated "Old Master" which can now be called a "New Master" and written out as NEWMAST. This process is shown in Figure 15-7.

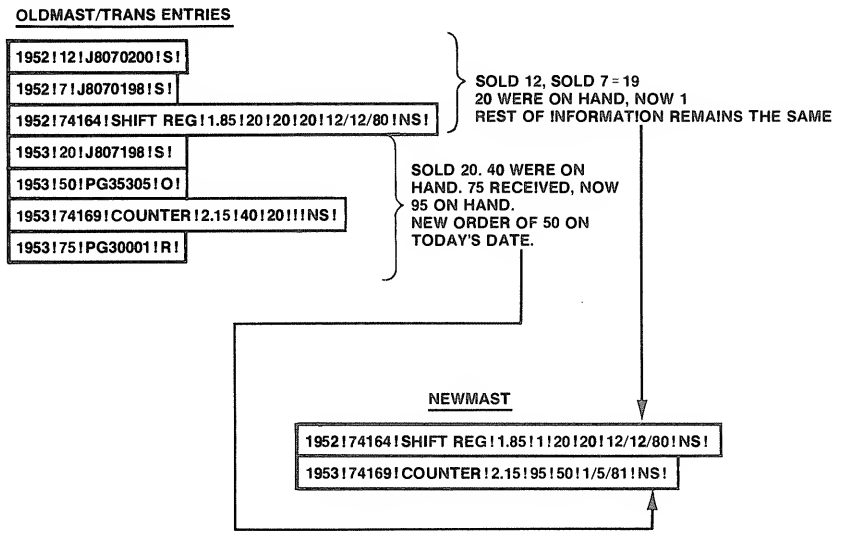


Figure 15-7. Creation of NEWMAST

The process of converting an Old Master into a New Master by a Transaction file should, of course, go hand in hand with disk "backups." It is only prudent to save each days TRANS file on a separate disk, along with the Old Master. This way a complete record is maintained of all stages of the inventory on a daily basis. Should an Act of God strike (such as an inventory clerk moving to Poughkeepsie), the inventory field can be reconstructed from backups.

A manual file maintenance procedure after the conversion saves OLDMAST and TRANS on a backup diskette, and RENAMES NEWMAST to OLDMAST for the next days transactions. This process is shown in Figure 15-8.

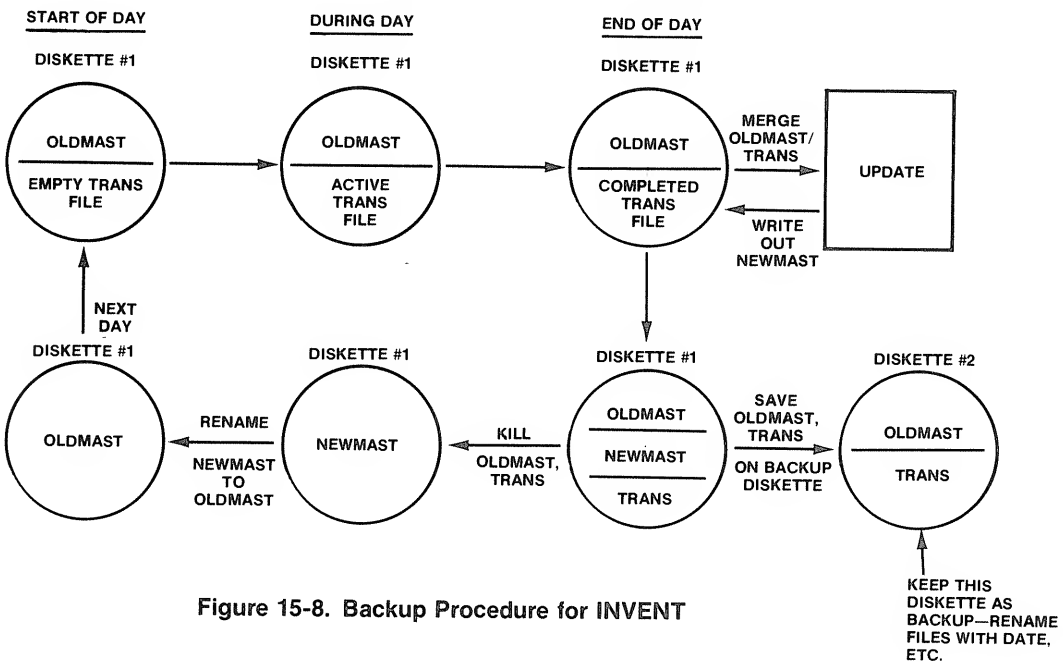


Figure 15-8. Backup Procedure for INVENT

What about the numeric processing problem? How can we convert the **string** data in the OLDMAST and TRANS to numeric values, adjust the OLDMAST value, and save it as string data once again? This can easily be done by use of the VAL and STR$ functions.

VAL takes a string value that represents numeric data and changes it to numeric variables. If the string is in ZW$(5), for example, the string would be converted to a numeric variable AA by:

```
1000 AA=VAL(ZW$(5))
```

Going the other way around, a numeric variable can be converted to a string variable by STR$:

```
2000 ZW$(5)=STR$(AA)
```

As long as the strings contain digits, a possible decimal point and no extraneous characters, there should be no problems in switching from string to numeric and back again. Once the switch has been made to numeric, arithmetic operations (such as adds, subtracts, multiplies, and divides) can be easily done. When the final result is obtained, it can be reconverted to a string value to be stored in an entry field. Here's an example of an add:

```
1000 AA$="234"
1010 AB$="567"
1020 BB=VAL(AA$)
1030 CC=VAL(AB$)
1040 DD=BB+CC
1050 AC$=STR$(DD)
1060 PRINT AA$,AB$,BB,CC,DD,AC$
1070 STOP
```

The display would be "234 567 234 567 801 801" when the string and numeric variables were printed.

Having described how we are going to handle the general approach in INVENT, let's generate a design specification. Note that although the design spec is another example of "top-down" design, we really did quite a bit of thinking about the actual structure and operation of the program before writing the design spec.

INVENT Design Spec

The design spec is shown in Figure 15-9.

INVENT Design Specification

OVERVIEW:

INVENT is an inventory control system that handles updating of an inventory file by daily transactions. The inventory file contains records that describe an inventory of parts by part number, description, number on hand, and other parameters. The daily transactions include processing of sales of parts, ordering of new parts to replenish the inventory, and receipt of parts for restocking.

INVENT may be used to

- Build and maintain a master file of parts
- Keep track of the stock of parts on a daily (or other) basis
- Keep track of the current orders for parts including number ordered and date of order
- Generate a report of the current parts inventory

LOADING INVENT:
To load INVENT, first load BASIC. If you are using the Model II, specify BASIC -F:1 to allow you to have at least one disk buffer. Next, load INVENT from disk by RUN INVENT. INVENT should start execution, and you should see the display shown in Figure 15-9-1. Now enter the model number of your system, 1 for Model I, 2 for Model II, or 3 for Model III. INVENT will continue initialization procedures as shown by the activity display in the upper right hand corner of the screen. There will be no display of the 1, 2, or 3 digit.

After initialization, INVENT will ask for the current date and will then display a menu of items as shown in Figure 15-9-2.

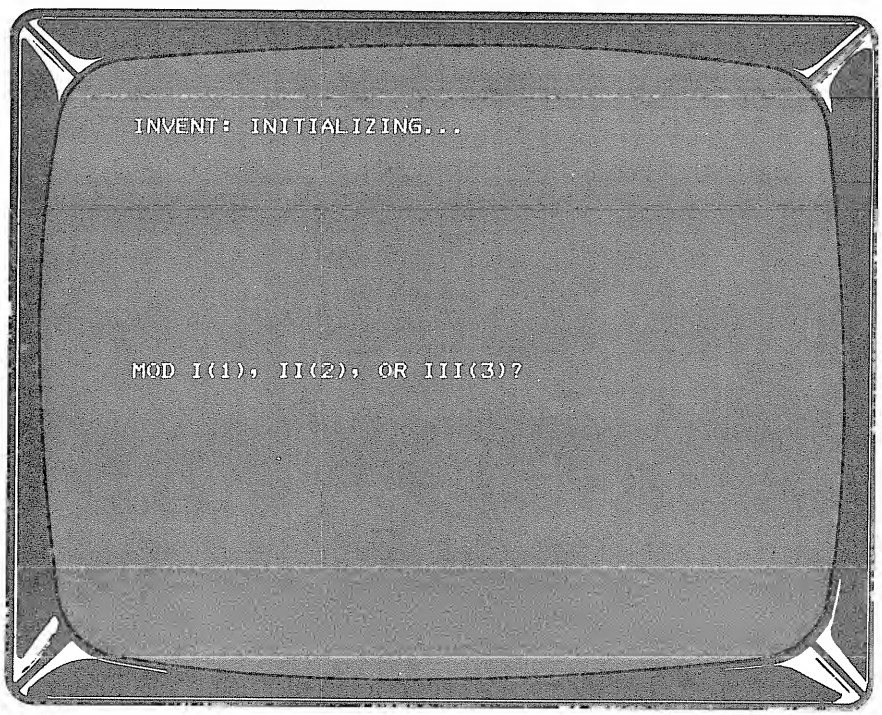


Figure 15-9-1. Loading INVENT

PROGRAM FUNCTIONS:

The program functions for INVENT are shown in the menu. The first three menu items pertain to "inventory master file" maintenance. The next four pertain to "transaction file" records, and the last two are used to update the master file and to provide an inventory report, respectively.

Inventory Master File

The inventory master file is a collection of records describing every part in the inventory. Each part is described by one "record" in the file. Each record is made up of eight fields that describe the part or inventory status of the part.

The fields are:

- Internal part number. A four digit number that defines your internal part number. This is the primary way parts are referenced.
- Vendor part number. A 12-character part number that is the vendor's part number.

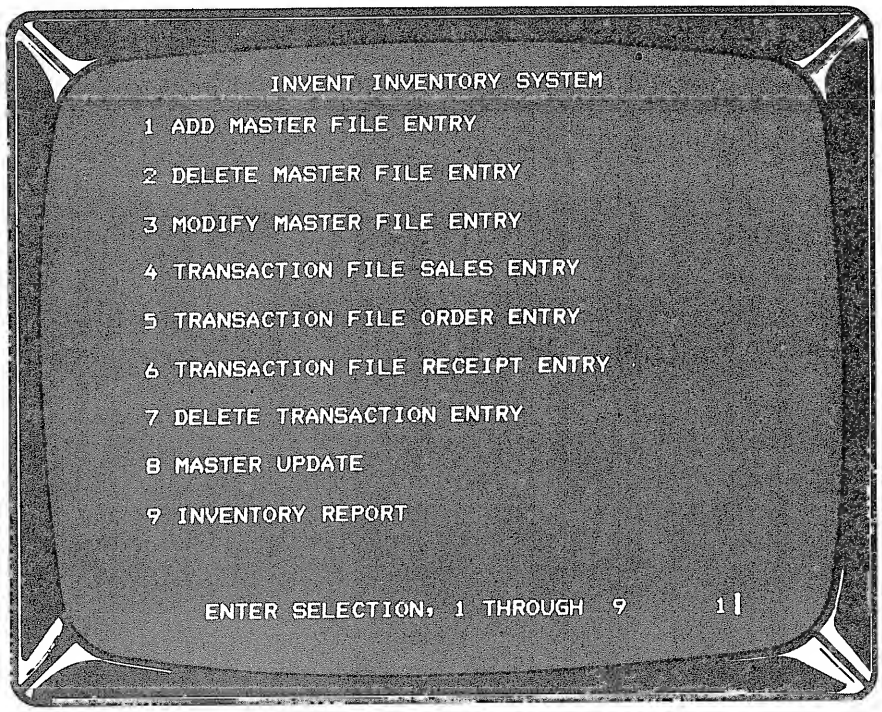


Figure 15-9-2. INVENT Menu

- Part description. A 15-character description of the part.
- List price. A six-digit list price of the part. The last three characters of this must be ".xx" where xx is the number of cents in the price.
- Number on hand. The actual number on hand in the inventory.
- Desired stock level. The number which optimally should be in stock.
- Number on order. The number on order, if any.
- Date of order. The date of order, if any.
- Vendor. A two-character code that describes the vendor.

The inventory master file is used as a running record of the inventory. Typically, the master file will be updated daily by the transaction file to produce a new master file for the next day's transactions.

Transaction File

The transaction file is a daily record of transactions that affect the inventory. There are three types of transactions: sales (parts are sold and removed from the inventory), orders (parts are reordered to restock the inventory), and receipts (parts are received and are placed in inventory).

The transaction file represents a daily record of these transactions. Sales, orders, and receipts are entered as they occur. At the end of the day, the transaction file is used to update the master inventory file to create a new master file for the next day.

A sales entry for the transaction file contains:

- Part number (your internal part number corresponding to the part number in the master file)
- The # shipped or sold
- Your invoice number
- An "S" for sales

An order entry for the transaction file contains:

- Part number (your internal part number corresponding to the part number in the master file)
- The number ordered
- Your purchase order number
- An "O" for order

A receipt entry for the transaction file contains:

- Part number (your internal part number corresponding to the part number in the master file)
- The number received and restocked
- The original purchase order number
- An "R" for receipt

Updating

Normally only the transaction file is used during periods of business operation. The master file may be maintained by adding new parts, deleting parts, or changing parts descriptions or data, but this would normally be done "after hours." The transaction file and master file are "merged" together to create a new master file at the end of each day (or periodically). The old master file and day's transactions are saved as a "backup," while the "new" master file is used for the next update.

Master File Functions

Menu items 1, 2, and 3 are used to create and maintain the master file.

To add a new master file entry, or to create a master file, select item 1. The form shown in Figure 15-9-3 will appear. Enter the data for the nine fields shown on the form in the format indicated.

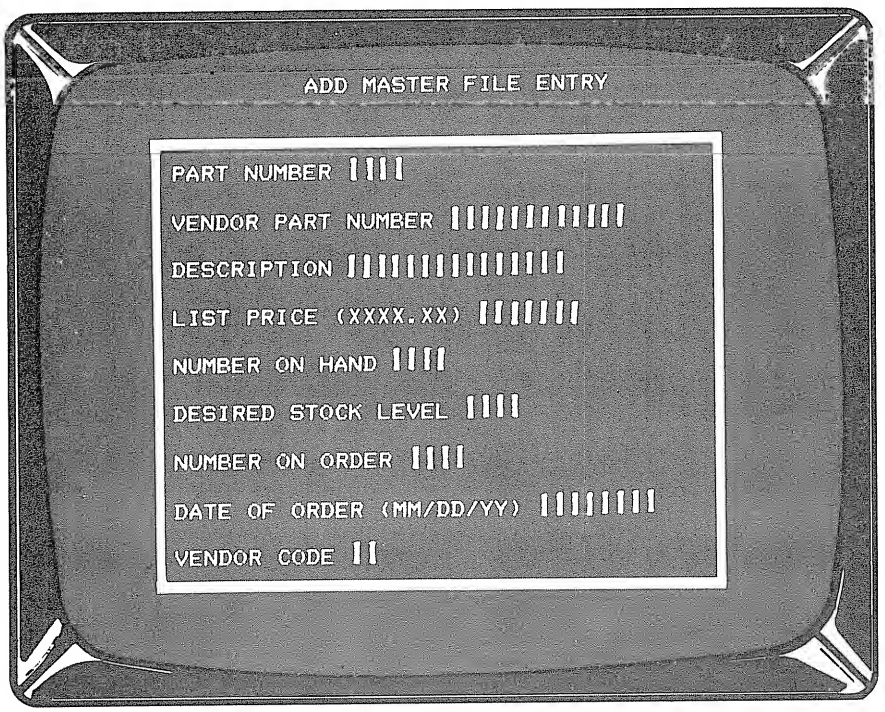


Figure 15-9-3. Add Master File Entry Form

To delete a master file entry, select menu item 2. The form shown in Figure 15-9-4 will appear. Enter the part number to be deleted. INVENT will search the file for the part number and display it on the screen in the format shown in Figure 15-9-5, along with the message DELETE?. Enter Y if you wish the item deleted, or N if you do not want to delete the item.

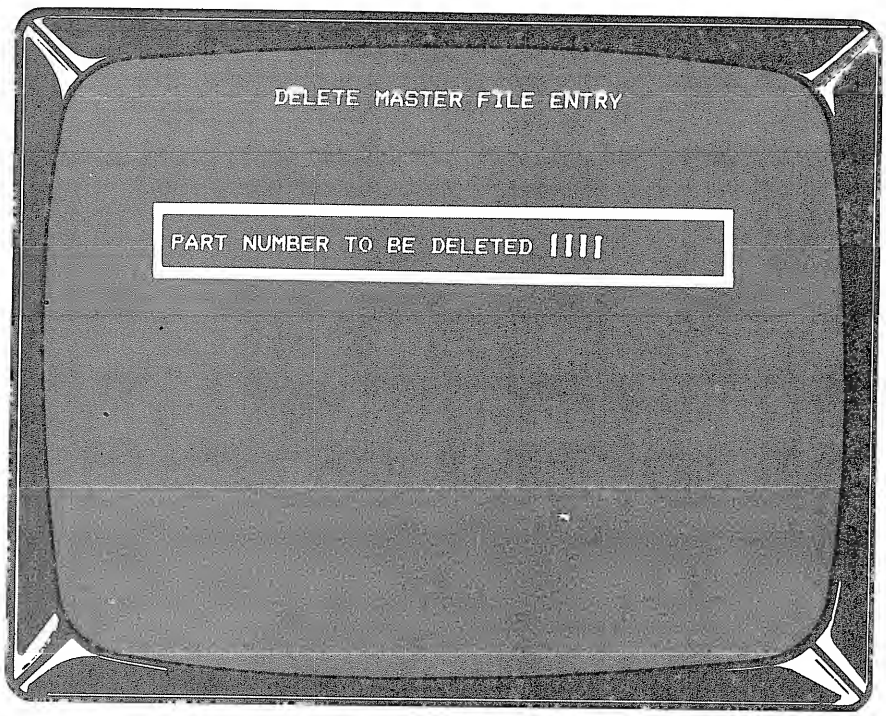


Figure 15-9-4. Delete Master File Entry Form

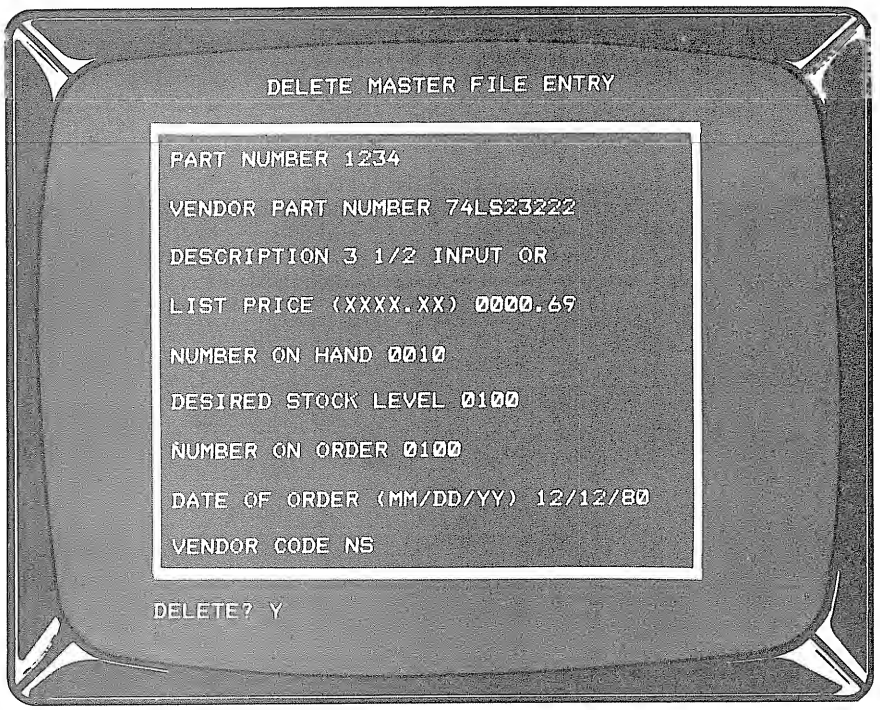


Figure 15-9-5. Typical Delete Entry Display

To modify a master file entry, select menu item number 3. The form shown in Figure 15-9-6 will appear. Enter the part number to be modified. INVENT will search the file for the part number and display it on the screen in the format shown in Figure 15-9-5, along with the message FIELD NUMBER TO MODIFY?. Enter the field number to be modified, and then enter the corrected data. Repeat as often as necessary. When the form is to your satisfaction, press ENTER and the corrected entry will be saved in the master file.

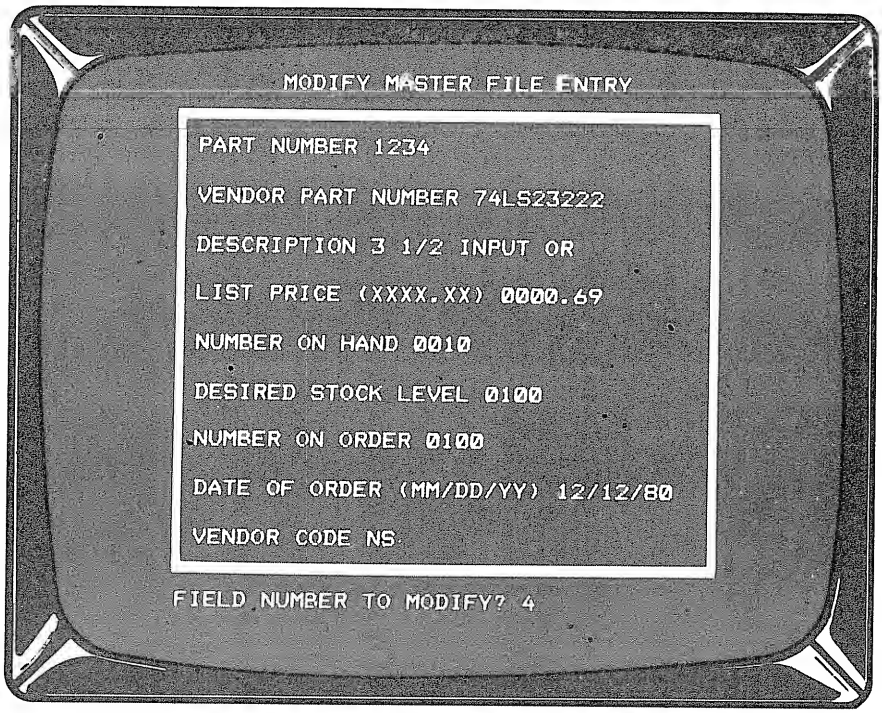


Figure 15-9-6. Modify Master File Entry Form

The functions above operate with a disk file called OLDMAST. This file is automatically loaded from disk when the first master file function is invoked and saved on disk when a menu function other than 1 through 3 is selected. To properly "close" the file, always select another menu function after the master file maintenance has been done.

Transaction File Functions

Menu items 4, 5, 6, and 7 are used to build a transaction file. When menu item 4 is selected, the form shown in Figure 15-9-7 will appear on the screen. Enter the fields for the form in the format indicated.

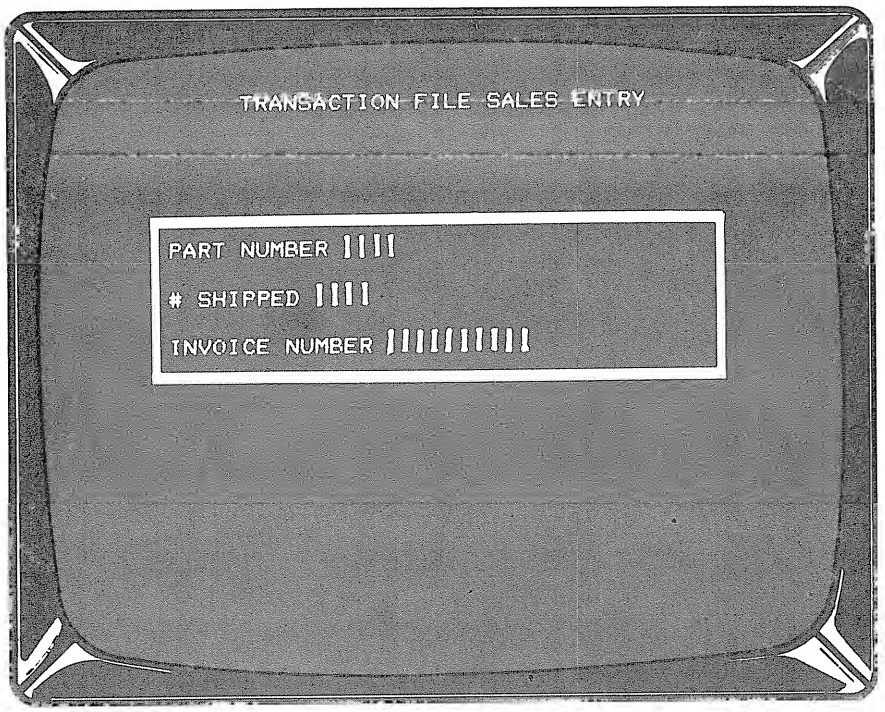


Figure 15-9-7. Sales Form

When menu item 5 is selected, the form shown in Figure 15-9-8 will appear on the screen. Enter the fields for the form in the format indicated.

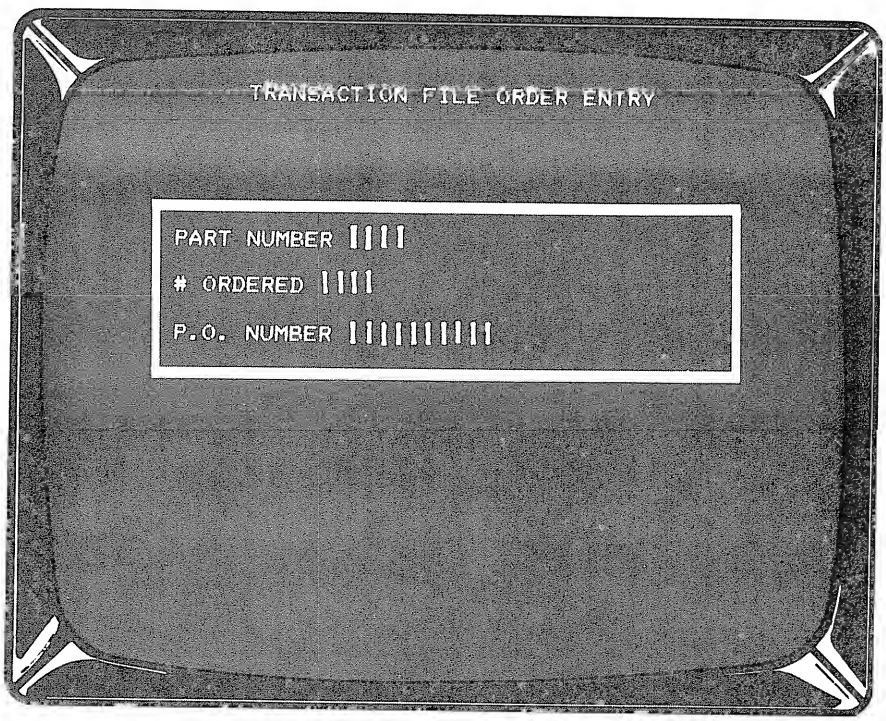


Figure 15-9-8. Order Form

When menu item 6 is selected, the form shown in Figure 15-9-9 will appear on the screen. Enter the fields for the form in the format indicated.

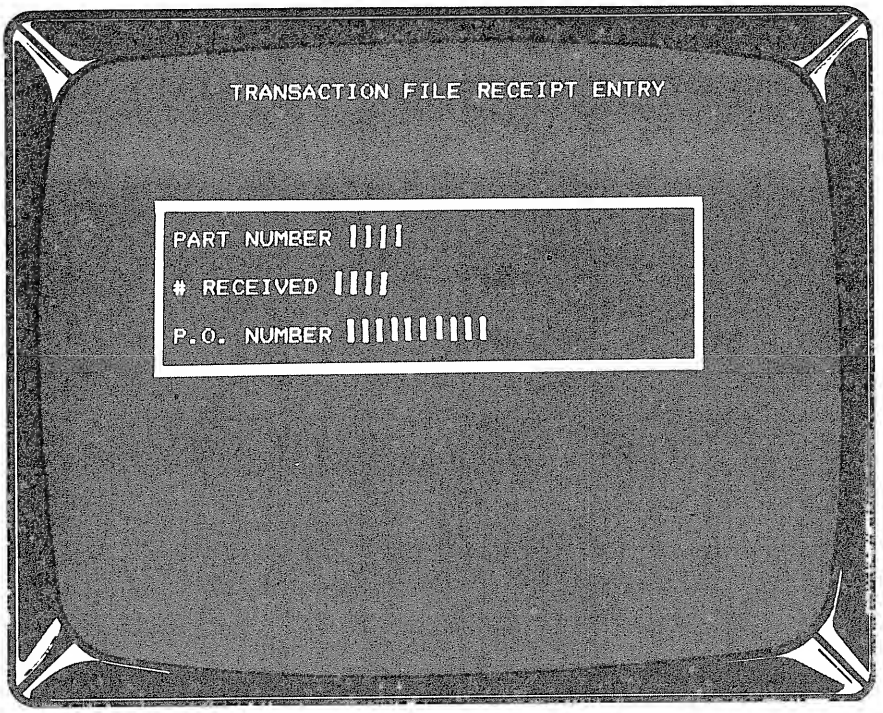


Figure 15-9-9. Receipt Form

Menu item 7 is a special item that will delete a previously entered transaction entry. When menu item 7 is selected, the form shown in Figure 15-9-10 will appear on the screen. Enter the approximate number of the transaction item. INVENT will retrieve that number entry and display it on the screen in the appropriate format, along with the question DELETE?. To delete the transaction, enter Y. To return to the menu, enter N or simply press ENTER. As deletions should be rare, this function allows a search for the transaction without embellishments. It may be necessary to perform the search several times to find the correct entry.

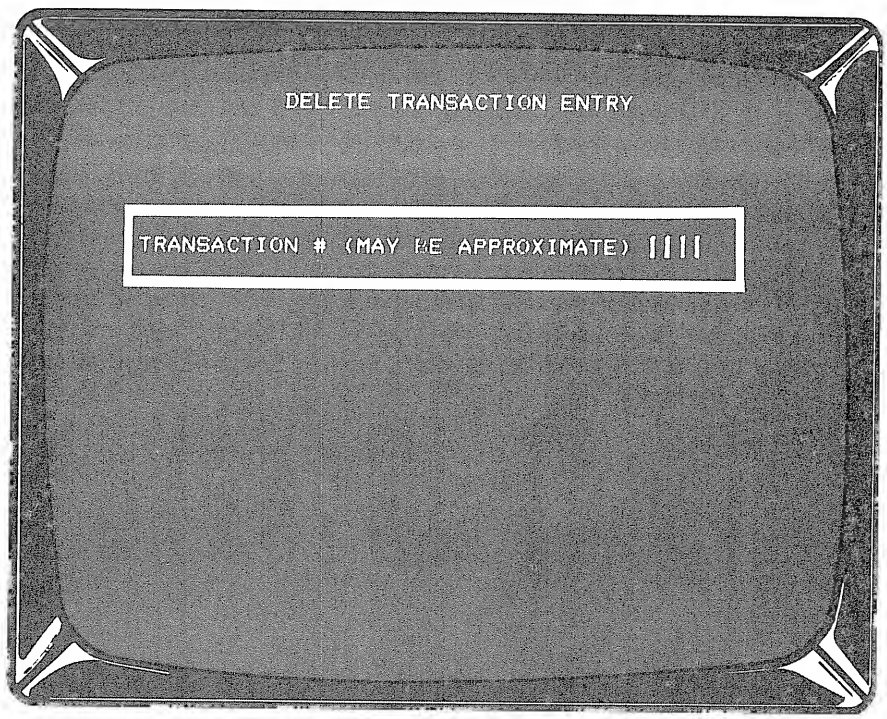


Figure 15-9-10. Delete Transaction Entry Form

Normally only menu items 4 through 7 will be used during the daily recording of transactions. These items work with the transaction file called TRANS. TRANS is automatically loaded when the first transaction item is selected, and saved when an item other than 4-7 is selected. Be certain to periodically save the current file in memory by selecting another menu item. An easy way to do this is to select item 9 (Report), and then to abort the operation by pressing CLEAR (Model I/III) or "up arrow" (Model II). The TRANS file in memory will be saved before the Report function is entered, and reloaded as it is aborted.

Update Function

Menu item 8 selects the update function. When this item is selected, the OLDMAST file will be loaded, followed by the TRANS. The two files will then be merged to produce one new file NEWMAST. INVENT will display messages at various stages of the operation. The steps are

- Save the TRANS file or OLDMAST file in memory if necessary.
- Load the TRANS file.

- Load (merge) the OLDMAST file.
- Update the OLDMAST file with TRANS data and save the new file on disk as NEWMAST.

At the end of the operation, three files are on disk — TRANS, OLDMAST, and NEWMAST. It is the user's responsibility to save the TRANS and OLDMAST files on another disk as backup and to RENAME the NEWMAST to OLDMAST for the next set of operations.

Report Operations

When menu item 9 is selected, a Report of the current OLDMAST on disk will be produced on the system line printer, and simultaneously displayed on the screen. After the selection, the message PRINTER READY? will appear. When the line printer is ready, enter Y. The report shown in Figure 15-9-11 will be produced from the OLDMAST file on disk.

```
                              INVENT INVENTORY REPORT
PART   VENDOR PART   DESCRIPTION        LIST PR  #OH   LVL  # OO     DATE     CD

4001   7452          EXPAND AND/OR        1.15    10    50     0              NS
4002   7453          EXPAND AND/OR/I      1.15     0    50   100  12/15/80 TI
4010   7455          AND/OR/I              .79    22    20    10  01/12/81 MO
4011   7474          FLIP-FLOP            1.25     7    50    50  01/12/81 TI
```

Figure 15-9-11. INVENT Report

Before INVENT produces the report, it will "purge" itself of any current activity. If it is working with the TRANS or OLDMAST files in memory, it will automatically save the file on disk and then load in the OLDMAST to produce the report. Report simply lists the current OLDMAST; it does not know whether OLDMAST has been updated by the current TRANS file.

Aborting Any Operation

Generally, any operation except saving a file on disk can be aborted by pressing the CLEAR key (Models I/III) or the "up arrow" key (Model II) to return to the menu of items.

General INVENT Design

Because this is such a large program compared to the other ones we've described in this book, we'll indicate in a general way how the INVENT functions may be implemented using the GPM modules. We'll supplement the description with flowcharts of some of the functions.

Master File Functions

The three master file functions of adding an entry, deleting an entry, and modifying an entry can easily be handled along the same lines as MAILLIST. The sequence of operations and forms would be very similar.

One difference is that the OLDMAST file would have to be automatically read into memory any time a master file function is entered from another function. A variable "flag" would keep track of the current file in memory. Similarly, going from a master file function to another function would automatically save the OLDMAST file on disk. These actions could be taken care of in the MAIN driver program as shown in Figure 15-10.

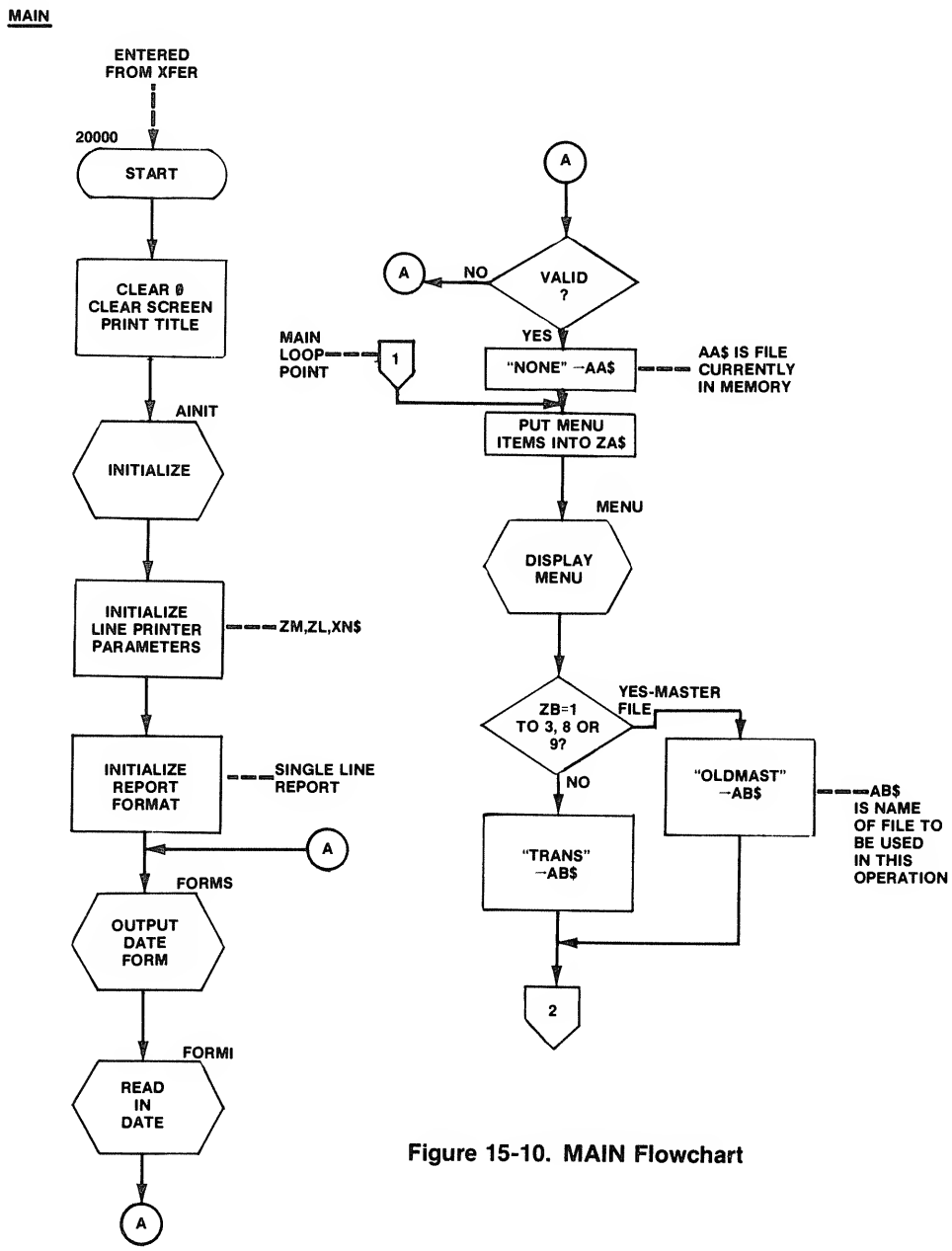


Figure 15-10. MAIN Flowchart

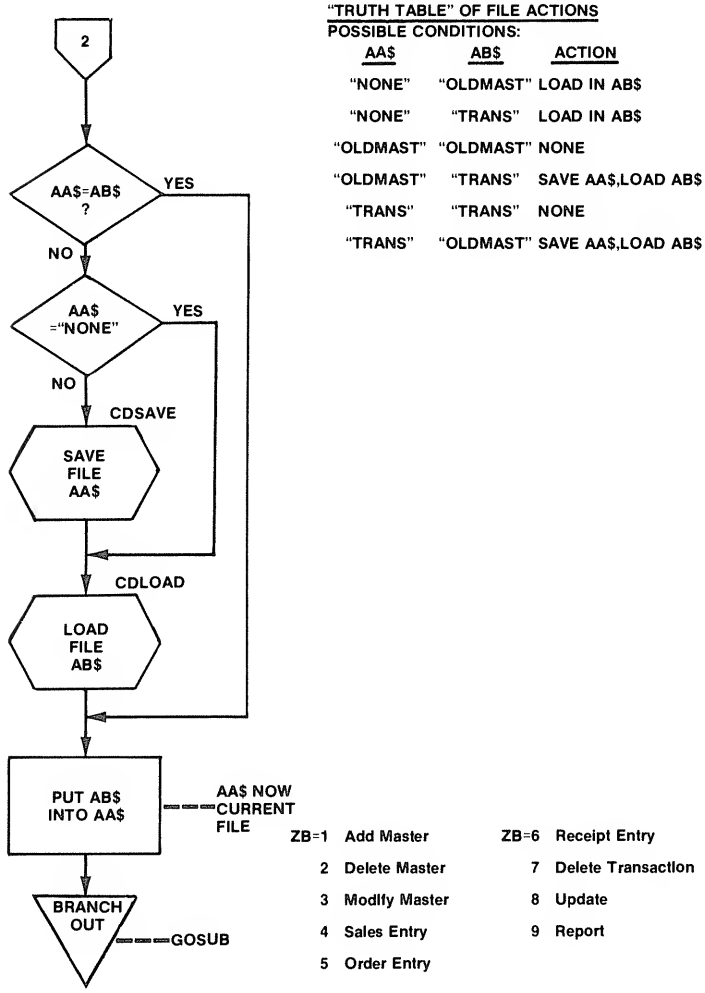


Figure 15-10. MAIN Flowchart (cont.)

Transaction File Functions

The flowchart in Figure 15-10 also shows the automatic load and save of the TRANS file depending upon the menu item selected. Any time a transaction file operation was being processed (most of the time), the TRANS file would be "resident." It would be saved only when another item was selected, and reloaded when a new transaction operation was started.

There are essentially two transaction file operations - adding an entry to the file and deleting an entry to the file. They can be handled in similar fashion to adding and deleting entries in MAILLIST. The O, R, and S codes should be automatically put into the transaction entry after the other data has been

entered. This can be done by putting them into ZW$(4) after the form has been filled in,as shown in Figure 15-11.

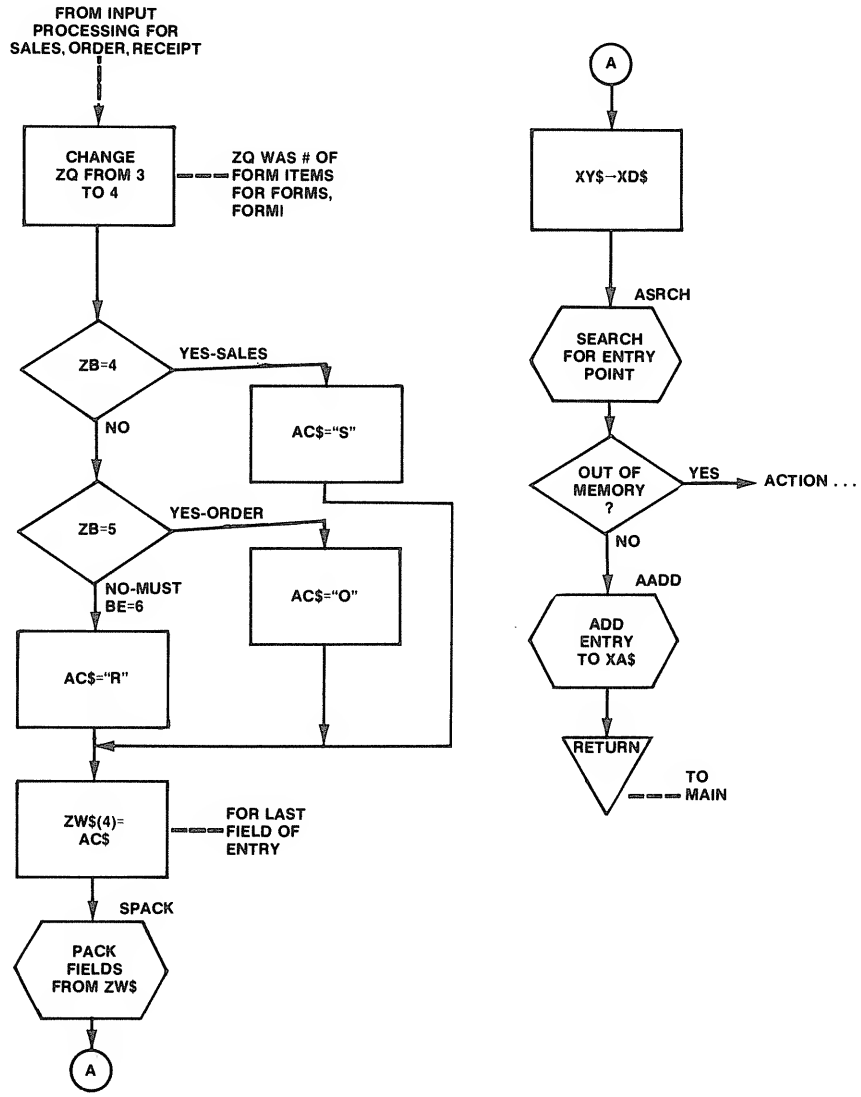


Figure 15-11. Transaction Entry Flowchart

The delete transaction entry can be handled much more simply than the delete in MAILLIST. Here, only an entry number is specified, and a call to FINDN can be done with the entry number, as shown in Figure 15-12.

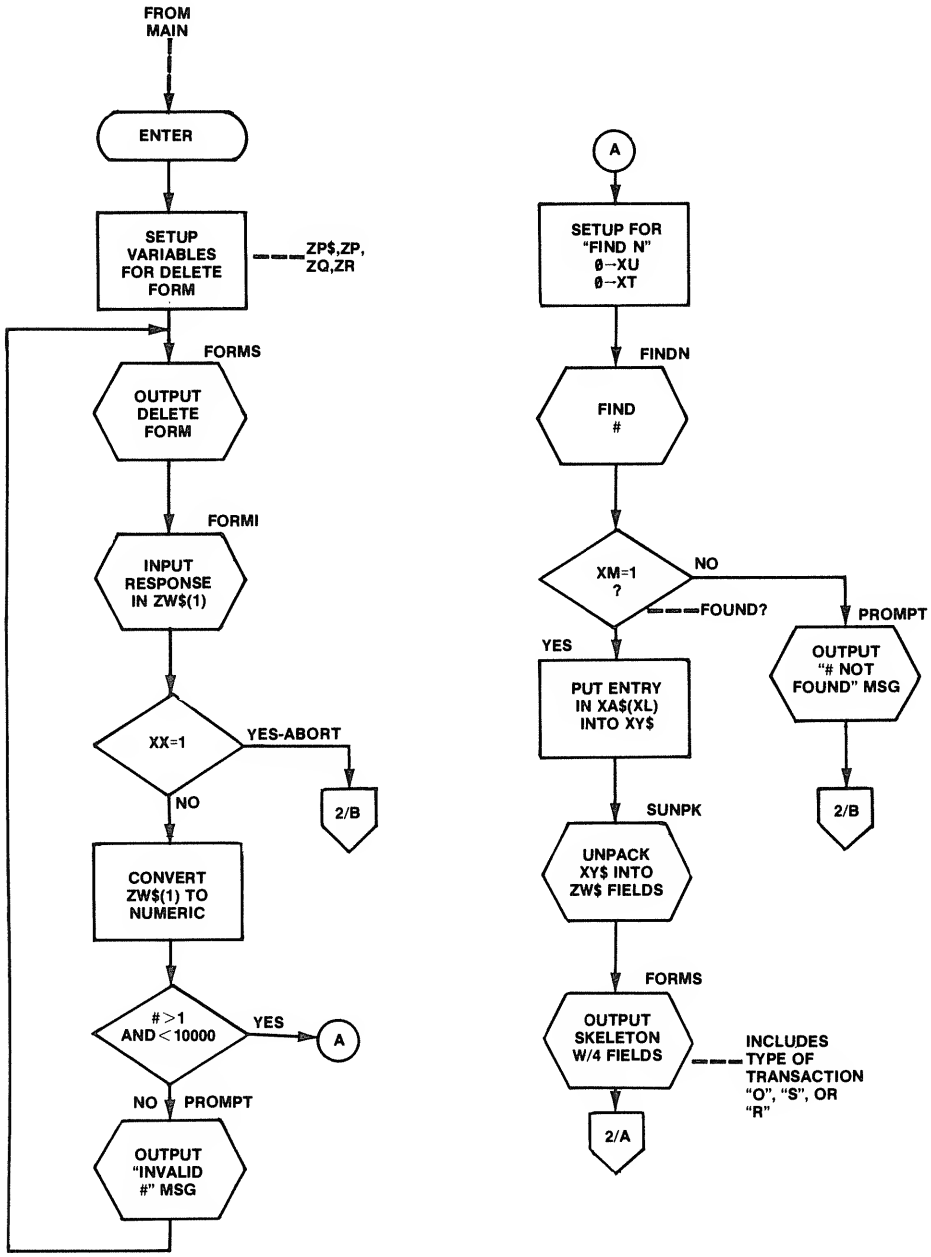


Figure 15-12. Transaction Deletion Flowchart

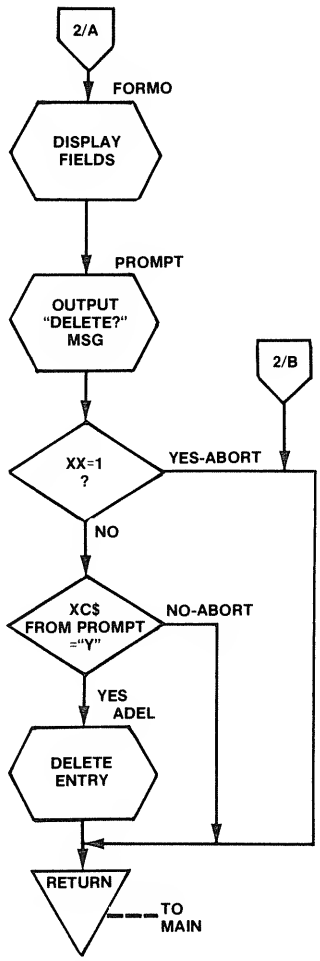


Figure 15-12. Transaction Deletion Flowchart (cont.)

Update Function

The update function is a new concept. It involves purging any TRANS file or OLDMAST file in memory, and then reading in OLDMAST from disk in a CDLOAD ""I"" (initialize) mode. Next, the TRANS file is loaded using CDLOAD in an ""M"" (merge) mode. The two files have now been merged in memory and are ordered by part number.

Now the TRANS file entries can be located for each OLDMAST entry. There may be no corresponding TRANS file entries for an OLDMAST entry or any number. As each TRANS entry is found, the OLDMAST entry is updated by the data in the TRANS entry. The TRANS entry is then deleted by a call to ADEL. When all

OLDMAST entries have been updated, only the OLDMAST entries remain, and the file can be written out as NEWMAST. The flowchart for this function is shown in Figure 15-13.

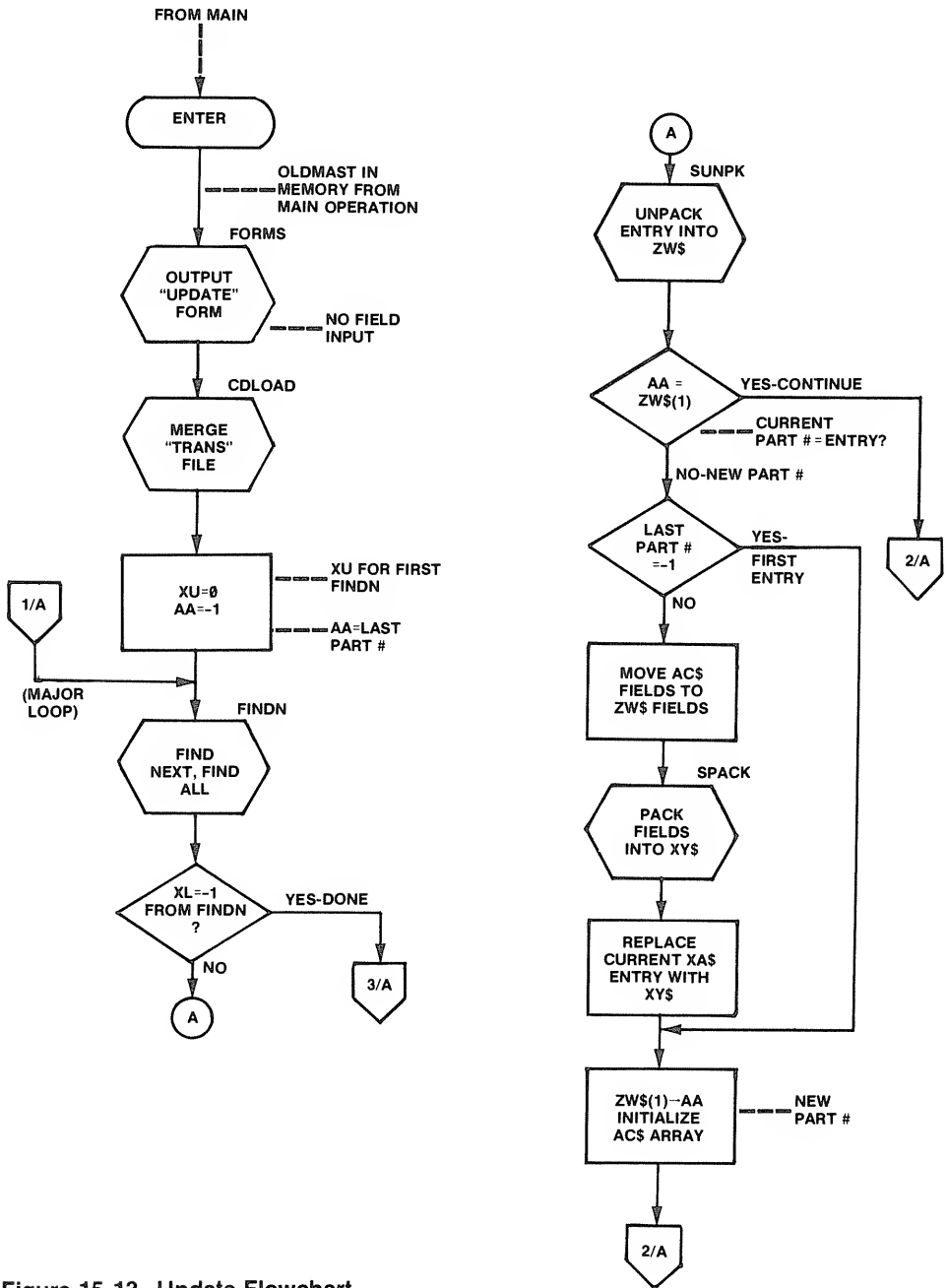


Figure 15-13. Update Flowchart

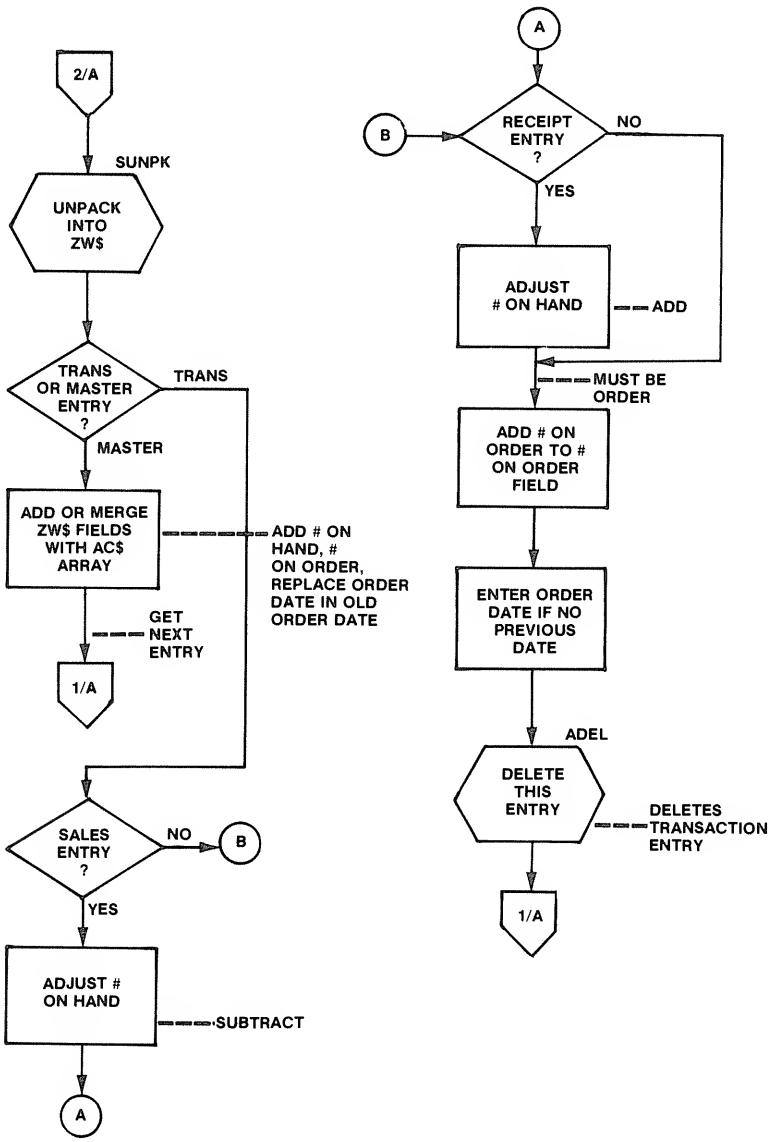


Figure 15-13. Update Flowchart (cont.)

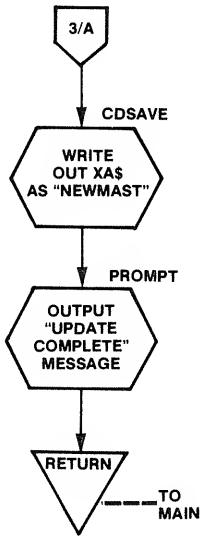


Figure 15-13. Update Flowchart (cont.)

Report Generation

The last function, report generation, is easy. (An old programming adage, however, says "**Nothing** is simple, **nothing** is easy" . . .) The REPORT format is defined in the XP array and will never change (it should be defined in MAIN).

The Report function purges the memory of the TRANS or OLDMAST file and then reads in OLDMAST. Each entry of OLDMAST, starting from entry 1, is then accessed by a call to FINDN. As each entry is found, it is unpacked into fields by SUNPK and output via REPORT. The flowchart for this operation is shown in Figure 15-14.

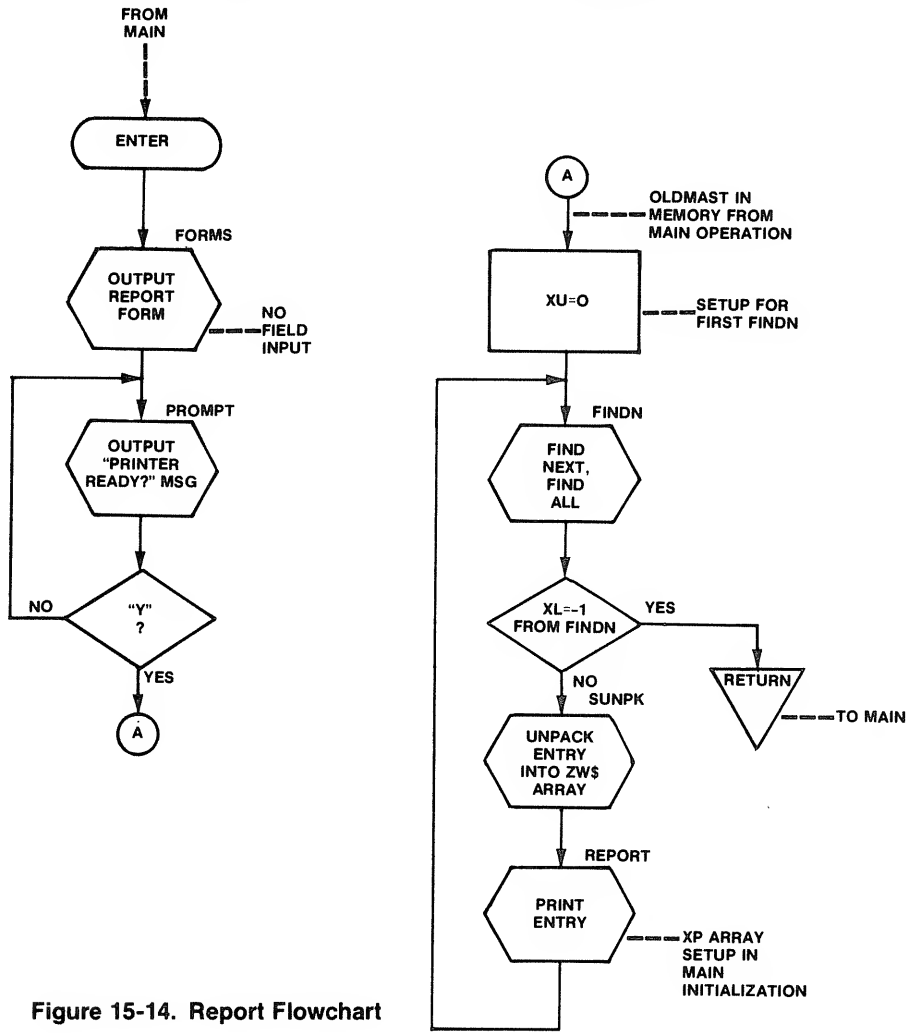


Figure 15-14. Report Flowchart

Special Problems in INVENT

There are no insoluble problems in INVENT. Probably the chief problem is that of memory limitations. Here again, the OLDMAST and TRANS files could be split up into several files based upon part number to facilitate handling larger inventories. Disk files could be also allocated on more than one drive.

There is no conflict in having one file with a different number of fields in each entry. The SUNPK routine will unpack any number of fields into the ZW$ array. As shown in the flowchart above for the update, another array must be used when two successive SUNPKs are done, for obvious reasons.

Using the GPM In Your Own Applications

The discussion above is not meant to be a definitive solution to an inventory program, but a suggested way to handle an inventory using the General Purpose Modules. At this point you should be able to construct your own solutions using the GPM file structure and modules. Perhaps you have learned enough in the examples of MAILLIST, KEYWORD, and INVENT to throw away the General Purpose Modules and construct your own approaches. If so, my feelings won't be hurt — as a matter of fact, I would be delighted! Happy applications programming!

Appendix I
General Purpose Modules

The following BASIC code contains all of the General Purpose Modules in "compressed" form. To use, key in the statements **exactly** as shown. We can't be responsible for applications in which **any** of the statements are modified; we will be held directly responsible for any errors that occur in the code shown.

Are there errors in this code? We have tried hard to debug the program, but there are invariably errors in any code. There will be two types of errors — invalid parameters passed to the GPM and logic errors.

The General Purpose Modules are not "fool-proof." They can be fooled easily by calling them with invalid parameters. You would not want to have subroutines that could not be fooled — they would check for every error condition and be horrendously slow. Your task, as an applications programmer, is to make certain that you call the GPM with correct parameters. These errors, then, are **your** responsibility.

If you are using the GPM exactly as shown and have found definite logic errors in the GPM please let us know and we will correct subsequent versions. These errors should be minimal.

```
120 GOTO20000
1000 GOTO1090'SSRCH
1090 IFLEN(XW$)>LEN(XZ$)THENSTOP
1100 XI=LEN(XZ$):XH=LEN(XW$)
1110 IFXW$="!"GOTO1150
1120 FORXW=1TOXI-XH+1:IFMID$(XZ$,XW,XH)=XW$GOTO1180
1130 NEXTXW
1140 GOTO1170
1150 FORXW=1TOXI:IFMID$(XZ$,XW,1)="!"GOTO1180
1160 NEXTXW
1170 XW=-1
1180 RETURN
1500 GOTO1630'LPDRIV
1630 IFZJ<>0GOTO1660
1640 ZJ=1
1650 ZK=0
1660 IFZM$=""GOTO1700
1670 LPRINTZM$
1680 ZK=ZK+1
1690 IFZK<>ZMGOTO1760
1700 FORZI=1TOZL-ZK
1710 LPRINT"                    "
1720 NEXTZI
1730 ZK=0
1740 IFZN$=""GOTO1760
1750 ZM$=ZN$:GOTO1500
1760 RETURN
2000 GOTO2110'INPUT
2110 XX=0
2120 IFZC<0ORZC>YA*YB-1THENSTOP
2130 IFZD<1ORZD>255THENSTOP
2140 PRINT@ZC,STRING$(ZD,CHR$(YJ));
2150 ZF$=""
2160 PRINT@ZC+LEN(ZF$),CHR$(YJ);
2170 ZE$=INKEY$
2180 IFZE$<>""GOTO2210
2190 PRINT@ZC+LEN(ZF$)," ";
2200 GOTO2160
2210 IFZE$>CHR$(YK)GOTO2300
2220 IFZE$=CHR$(13)GOTO2350
2230 IFZE$<>CHR$(YK)GOTO2250
2240 XX=1:GOTO2360
2250 IFZE$<>CHR$(8)GOTO2160
2260 IFZF$=""GOTO2290
2270 ZF$=LEFT$(ZF$,LEN(ZF$)-1)
2280 PRINT@ZC,ZF$+CHR$(YJ);
2290 GOTO2160
2300 IFLEN(ZF$)=ZDGOTO2350
2310 IFZE$=","THENZE$=";"
2320 ZF$=ZF$+ZE$
2330 PRINT@ZC,ZF$;
2340 GOTO2160
2350 IFZE=0THENZF=VAL(ZF$)
2360 RETURN
2500 GOTO2600'SUNPK
2600 ZQ=1:XW$="!":XZ$=XY$:ZI=LEN(XZ$)
```

```
2610 GOSUB1000:IFXW=-1GOTO2690
2620 ZW$(ZQ)=MID$(XZ$,1,XW-1)
2630 ZX(ZQ)=XW-1
2640 ZI=ZI-XW
2650 IFZI<1GOTO2690
2660 XZ$=MID$(XZ$,XW+1,ZI)
2670 ZQ=ZQ+1
2680 GOTO2610
2690 RETURN
5000 GOTO5140'ASRCH
5140 IFXA%(0)<>-1GOTO5180
5150 XJ=0:XK=1:XL=-1:XM=0:XS=0
5160 GOTO5370
5180 FORXK=1TOXQ
5190 IFXA%(XK)=-2GOTO5240
5200 NEXTXK
5210 XM=2
5220 GOTO5370
5240 XJ=0:XS=1
5250 XL=XA%(0)
5260 PRINT@YL,XS;"    ";
5270 IFXD$<>XA$(XL)GOTO5310
5290 XM=1
5300 GOTO5370
5310 IFXD$<XA$(XL)GOTO5360
5320 XJ=XL
5330 XL=XA%(XL)
5340 XS=XS+1
5350 IFXL<>-1GOTO5260
5360 XM=0
5370 RETURN
5500 GOTO5650'FINDN
5650 IFXU<>0GOTO5740
5660 IFXA%(0)=-1GOTO5770
5670 XU=1:XJ=0
5680 IFYS=0THENXL=XA%(0)ELSEXL=XB%(0)
5690 IFXU<>XSGOTO5720
5700 XM=1
5710 GOTO5780
5720 IFXT=1GOTO5770
5730 XU=XU+1
5740 XJ=XL:IFYS=0THENXL=XA%(XL)ELSEXL=XB%(XL)
5750 IFXL=-1GOTO5770
5760 GOTO5690
5770 XM=0
5780 PRINT@YL,XU;"    ";
5790 XU=XU+1
5800 RETURN
6000 GOTO6100'SECSRT
6100 FORXI=1TOXQ-3STEP4
6110 XB%(XI)=-2:XB%(XI+1)=-2:XB%(XI+2)=-2:XB%(XI+3)=-2
6120 PRINT@YL,XI;
6130 NEXTXI
6140 XB%(0)=-1
```

```
6150 IFXA%(0)=-1GOTO6350
6160 XK=XA%(0)
6180 PRINT@YL,XK;"    ";
6190 XZ$=XA$(XK)
6200 XW$="!"
6210 IFYT=1GOTO6260
6220 FORZI=1TOYT-1
6230 GOSUB1000
6240 XZ$=MID$(XZ$,XW+1,LEN(XZ$)-XW)
6250 NEXTZI
6260 GOSUB1000
6270 XD$=LEFT$(XZ$,XW-1)
6290 GOSUB6360
6300 XB%(XK)=XL
6310 XB%(XJ)=XK
6330 XK=XA%(XK)
6340 IFXK<>-1GOTO6180
6350 RETURN
6360 IFXB%(0)<>-1GOTO6390
6370 XJ=0:XL=-1
6380 GOTO6450
6390 XJ=0
6400 XL=XB%(0)
6410 XY$=XA$(XL):GOSUB2500
6420 IFXD$<ZW$(YT)GOTO6450
6430 XJ=XL:XL=XB%(XL)
6440 IFXL<>-1GOTO6410
6450 RETURN
6500 GOTO6600'SPACK
6600 XY$=""
6610 FORXI=1TOZQ
6620 XY$=XY$+ZW$(XI)+"!"
6630 NEXTXI
6640 RETURN
7000 GOTO7070'AADD
7070 XA$(XK)=XD$
7080 XA%(XK)=XL
7090 XA%(XJ)=XK
7100 RETURN
7500 GOTO7640'REPORT
7640 IFXP(0)<1THENSTOP
7650 ZM$=""
7660 FORXI=1TOXP(0)
7670 IFXP(XI)<>0GOTO7700
7680 IFZM$=""THENZM$="            "
7690 GOSUB1500:ZM$="":GOTO7770
7700 IFXP(XI)>0THENZM$=ZM$+ZW$(XP(XI))+" ":GOTO7770
7710 IFXP(XI)<-2THENZM$=ZM$+STRING$(-XP(XI)-LEN(ZM$)," "):GOTO7770
7720 IFXP(XI)<>-2GOTO7760
7730 ZM$=ZM$+STR$(XN)
7740 XN=XN+1
7750 GOTO7770
7760 ZM$="":GOSUB1500
7770 NEXTXI
7780 RETURN
```

```
8000 GOTO8130'FORMS
8130 IFZP<10ORZP>YA-4THENSTOP
8140 IFZ@<10RZ@>12THENSTOP
8150 CLS
8160 PRINTCHR$(2);
8170 PRINT@YA/2-LEN(ZP$(0))/2,ZP$(0);
8180 ZI=(YA-ZP)/2
8190 PRINT@ZI+YC,STRING$(ZP,CHR$(YF));
8200 PRINT@ZI+YD+YA*Z@,STRING$(ZP,CHR$(YG));
8210 FORZH=ZI+YDTOZI+YD+YA*(Z@-1)STEPYA
8220 PRINT@ZH,CHR$(YH);
8230 PRINT@ZH+ZP-1,CHR$(YI);
8240 NEXTZH
8250 FORZH=1TOZ@
8260 PRINT@ZI+1+YC+YA*ZH,ZP$(ZH);" ";STRING$(ZR(ZH),CHR$(YJ));
8270 ZS(ZH)=ZI+1+YC+YA*ZH+LEN(ZP$(ZH))+1
8280 NEXTZH
8290 RETURN
8500 GOTO8570'FORMI
8570 FORZI=1TOZ@
8580 ZC=ZS(ZI):ZD=ZR(ZI):ZE=1
8590 GOSUB2000:IFXX=1GOTO8620
8600 ZW$(ZI)=ZF$
8610 NEXTZI
8620 RETURN
9000 GOTO9080'FORMO
9080 FORZI=1TOZ@
9090 PRINT@ZS(ZI),ZW$(ZI);STRING$(ZR(ZI)-LEN(ZW$(ZI))," ");
9100 NEXTZI
9110 RETURN
9500 GOTO9570'ADEL
9570 XK=XA%(XL)
9580 XA%(XJ)=XK
9590 XA%(XL)=-2
9600 XA$(XL)="*"
9610 RETURN
10000 GOTO10140'MENU
10140 IFZA<10RZA>10THENSTOP
10150 CLS
10160 PRINTCHR$(2);
10170 ZI=LEN(ZA$(0))
10180 PRINT@YA/2-ZI/2,ZA$(0);
10190 FORZI=1TOZA
10200 PRINT@ZI*YA+YA+10,ZI;ZA$(ZI);
10210 NEXTZI
10220 PRINT@ZI*YA+YC+15,"ENTER SELECTION, 1 THROUGH ";ZA;
10230 ZC=(ZI*YA+YC+50):ZD=2:ZE=0:GOSUB2000
10240 IFZF<10RZF>ZAGOTO10220
10250 ZB=ZF
10260 RETURN
10500 GOTO10610'AINIT
10610 CLEARINT(MEM*.85)'***.85MAYBEADJUSTED***
```

```
10620 PRINTCHR$(2);
10630 DEFINTX,Y,Z
10640 XJ=0:XK=0:XL=0:XS=0:XU=0:XQ=0:YS=0:XT=0:XI=0:XH=0
10650 ZI=0:ZH=0:XW$="":XD$="":XZ$="":XY$=""
10660 XQ=INT(MEM/52)*4
10670 YE=320:YL=55
10680 DIMXA%(XQ):DIMXA$(XQ):DIMXP(20):DIMZW$(20):DIMXB%(XQ)
10690 DIMZA$(11):DIMZP$(13):DIMZZ(11):DIMZZ$(11)
10700 FORXI=1TOXQ-3STEP4
10710 XA%(XI)=-2:XA%(XI+1)=-2:XA%(XI+2)=-2:XA%(XI+3)=-2
10720 XA$(XI)="*":XA$(XI+1)="*":XA$(XI+2)="*":XA$(XI+3)="*"
10730 PRINT@YL,XI;
10740 NEXTXI
10750 PRINT@YL,"      ";
10760 XA%(0)=-1:XA$(0)="*"
10770 PRINT@320,"MOD I(1), II(2), OR III(3)?"
10780 XI$=INKEY$:IFXI$=""GOTO10780
10790 XC=VAL(XI$)
10800 IFXC<1ORXC>3GOTO10770
10810 IFXC=2THENYA=80ELSEYA=64
10820 IFXC=2THENYB=24ELSEYB=16
10830 YC=YA*2:YD=YA*3:YE=YA*(YB-1)+10
10840 IFXC<>2GOTO10870
10850 YF=150:YG=150:YH=148:YI=148:YJ=170:YK=30:YL=71
10860 GOTO10880
10870 YF=176:YG=131:YH=149:YI=170:YJ=138:YK=31:YL=55
10880 ONERRORGOTO11500
10890 GOTO20100'***CHANGETHISFORYOURSYSTEM***
11000 GOTO11110'PROMPT
11110 XX=0
11120 PRINT@YE,XB$+"                    ";
11130 IFXB=3GOTO11280
11140 XC$=""
11150 XI$=INKEY$:IFXI$=""GOTO11150
11160 IFXI$>CHR$(YK)GOTO11200
11170 IFXI$<>CHR$(YK)GOTO11190
11180 XX=1:GOTO11290
11190 IFXI$=CHR$(13)GOTO11230
11200 XC$=XC$+XI$
11210 PRINT@YE+LEN(XB$)+1,XC$;
11220 GOTO11150
11230 IFXB=0THENXC=VAL(XC$)
11240 IFXB<>2GOTO11290
11250 IFXC$<>"YES"ANDXC$<>"Y"ANDXC$<>"NO"ANDXC$<>"N"GOTO11120
11260 XC$=LEFT$(XC$,1)
11270 GOTO11290
11280 FORXI=1TO900:NEXTXI
11290 RETURN
11500 GOTO11640'ERROR
11640 IFZZ(0)=0GOTO11710
11650 FORXF=1TOZZ(0)
11660 IFXA=80GOTO11690
11670 IFERR/2=ZZ(XF)GOTO11740
11680 GOTO11700
11690 IFERR=ZZ(XF)GOTO11740
```

```
11700 NEXTXF
11710 XB$="CATASTROPHIC SYSTEM ERROR":XB=3:GOSUB11000
11720 ONERRORGOTO0
11730 RESUME
11740 XB$=ZZ$(XF):XB=3:GOSUB11000
11750 XX=1
11760 RESUMENEXT
12000 GOTO12110'CDLOAD
12110 IFZW$(3)="M"GOTO12200
12130 FORXI=1TOX0-3STEP4
12140 XA%(XI)=-2:XA%(XI+1)=-2:XA%(XI+2)=-2:XA%(XI+3)=-2
12150 XA$(XI)="*":XA$(XI+1)="*":XA$(XI+2)="*":XA$(XI+3)="*"
12160 PRINT@YL,XI;
12170 NEXTXI
12180 PRINT@YL,"      ";
12190 XA%(0)=-1:XA$(0)="*"
12200 IFZW$(1)<>"D"GOTO12260
12210 ZZ(0)=1:ZZ(1)=53:ZZ$(1)="FILE NOT FOUND"
12220 OPEN"I",1,ZW$(2)
12230 ZZ(0)=0
12240 IFXX=1GOTO12390
12260 XI=0
12270 IFZW$(1)="C"THENINPUT#-1,XY$ELSEINPUT#1,XY$
12280 IFXY$="*"THENGOTO12380
12290 IFZW$(3)="M"GOTO12350
12310 XA%(XI)=XI+1:XA$(XI+1)=XY$:XI=XI+1:XA%(XI)=-1
12320 PRINT@YL,XI;
12330 GOTO12270
12350 XD$=XY$:GOSUB5000
12360 GOSUB7000
12370 GOTO12270
12380 IFZW$(1)="D"THENCLOSE1
12390 RETURN
12500 GOTO12580'CDSAVE
12580 IFZW$(1)="C"GOTO12600
12590 OPEN"O",1,ZW$(2)
12600 XU=0
12610 XS=-1:XT=1:GOSUB5500
12620 IFXL=-1GOTO12650
12630 IFZW$(1)="D"THENPRINT#1,XA$(XL)ELSEPRINT#-1,XA$(XL)
12640 GOTO12610
12650 IFZW$(1)="D"THENPRINT#1,"*"ELSEPRINT#-1,"*"
12660 IFZW$(1)="D"THENCLOSE1
12670 RETURN
```

Appendix II
MAILLIST Program

The following code is a "compressed" version of the MAILLIST program. Key in the code together with the General Purpose Module code of Appendix I to form one total MAILLIST program. The completed code will occupy approximately 11,429 bytes.

Here again, we'd like to hear about hard logic errors that are not errors caused by modified code or invalid parameters passed to the General Purpose Modules.

```
20000 GOTO20010'MAIN
20010 CLEAR0
20060 CLS:PRINT@0,"MAIL LIST: INITIALIZING...";
20080 GOTO10500
20100 ZM=-1:ZL=66:XN$="                                                MAIL LIST"
20120 XP(0)=13:XP(1)=0:XP(2)=2:XP(3)=1:XP(4)=0:XP(5)=3
20130 XP(6)=0:XP(7)=4:XP(8)=0:XP(9)=5:XP(10)=6:XP(11)=7
20140 XP(12)=0:XP(13)=0
20160 ZA=8:ZA$(0)="MAIL LIST":ZA$(1)="ADD ENTRY TO FILE"
20170 ZA$(2)="MODIFY OLD ENTRY":ZA$(3)="DELETE ENTRY"
20180 ZA$(4)="DISPLAY/PRINT FILE":ZA$(5)="SEARCH FILE"
20190 ZA$(6)="NEW SORT":ZA$(7)="LOAD FILE":ZA$(8)="SAVE FILE"
20200 GOSUB10000
20230 IFZB<40RZB=7THENYT=0
20240 ONZBGOSUB20500,21000,21500,22000,23000,23500,24000,24500
20250 GOTO20160
20500 GOTO20550'MFADD
20550 ZP$(0)="ADD ENTRY":GOSUB29000
20570 GOSUB8500:IFXX=1GOTO20680
20590 GOSUB6500
20610 XD$=XY$:GOSUB5000
20630 IFXM<>2GOTO20670
20640 XB=3:XB$="OUT OF MEMORY":GOSUB11000
20650 GOTO20680
20670 GOSUB7000
20680 RETURN
21000 GOTO21050'MFMOD
21050 ZP$(0)="MODIFY ENTRY"
21080 GOTO21560
21100 XB$="MODIFY YES OR NO":XB=1:GOSUB11000:IFXX=1GOTO21240
21110 IFXC$="N"GOTO21240
21130 GOSUB9500
21140 XB$="FIELD # TO MODIFY":XB=0:GOSUB11000:IFXX=1GOTO21240
21150 IFXC=0GOTO21230
21160 IFXC<1GOTO21140
21170 GOSUB9000
21190 ZC=ZS(XC):ZD=ZR(XC):ZE=1:GOSUB2000:IFXX=1GOTO21240
21200 ZW$(XC)=ZF$
21210 GOTO21140
21230 GOSUB20590
21240 RETURN
21500 GOTO21550'MFDEL
21550 ZP$(0)="DELETE ENTRY"
21560 ZP=55:ZQ=2:ZP$(1)="ENTRY # (ENTER IF NOT KNOWN)"
21570 ZP$(2)="NAME1 STRING (ENTER IF NOT KNOWN)"
21580 ZR(1)=3:ZR(2)=15:GOSUB8000
21600 GOSUB8500:IFXX=1GOTO21900
21610 IFZW$(1)=""GOTO21650
21620 XS=VAL(ZW$(1))
21630 IFXS=0GOTO21650
21640 GOTO21750
21650 IFZW$(2)<>""GOTO21690
21660 XB=3:XB$="REENTER VALID # OR NAME":GOSUB11000
```

```
21670 GOTO21560
21690 XD$=ZW$(2):GOSUB5000
21710 IFXM<>0ORXL<>-1GOTO21790
21720 XB=3:XB$="# OR NAME NOT FOUND":GOSUB11000
21730 GOTO21900
21750 XU=0:XT=0:GOSUB5500
21770 IFXM=0GOTO21720
21790 XY$=XA$(XL):GOSUB2500
21810 GOSUB29000
21830 GOSUB9000
21850 IFZB=2GOTO21100
21860 XB$="DELETE YES OR NO":XB=1:GOSUB11000:IFXX=1GOTO21900
21870 IFXC$="N"GOTO21900
21890 GOSUB9500
21900 RETURN
22000 GOTO22050'MFDISP
22050 ZP=55:ZQ=1:ZP$(0)="DISPLAY/PRINT"
22060 ZP$(1)="PRIMARY (P) OR SECONDARY (S) KEY?"
22070 ZR(1)=1:GOSUB8000
22080 GOSUB8500:IFXX=1GOTO22950
22090 IFZW$(1)<>"P"ANDZW$(1)<>"S"GOTO22050
22100 YS=0:IFZW$(1)="P"GOTO22220
22120 IFYT<>0GOTO22150
22130 XB=3:XB$="NEVER SORTED OR MODIFIED-RESORT!":GOSUB11000
22140 GOTO22050
22150 YS=1:ZP=55:ZQ=2:ZP$(0)="SECONDARY DISPLAY/PRINT"
22160 ZP$(1)="START # (ENTER IF NOT KNOWN)":ZP$(2)="END # (ENTER IF NOT KNOWN)"
22170 ZR(1)=3:ZR(2)=3:GOSUB8000
22190 GOSUB8500:IFXX=1GOTO22950
22200 ZW$(3)=ZW$(2):ZW$(2)="":ZW$(4)=""
22210 GOTO22310
22220 ZP=55:ZQ=4:ZP$(0)="PRIMARY DISPLAY/PRINT"
22230 ZP$(1)="START# (ENTER IF NOT KNOWN)"
22240 ZP$(2)="START ENTRY (ENTER IF NOT KNOWN)"
22250 ZP$(3)="END# (ENTER IF NOT KNOWN)"
22260 ZP$(4)="END ENTRY (ENTER IF NOT KNOWN"
22270 ZR(1)=3:ZR(2)=15:ZR(3)=3:ZR(4)=15:GOSUB8000
22290 GOSUB8500:IFXX=1GOTO22950
22310 IFZW$(1)=""GOTO22370
22320 XU=0:XS=VAL(ZW$(1)):XT=0:GOSUB5500
22330 IFXM<>0GOTO22420
22340 XB$="START # NOT FOUND"
22350 XB=3:GOSUB11000
22360 GOTO22050
22370 IFZW$(2)<>""GOTO22390
22380 XB$="NO START # OR STRING":GOTO22350
22390 XD$=ZW$(2):GOSUB5000
22400 IFXM=0ANDXL=-1THENGOTO22410ELSEGOTO22420
22410 XB$="START STRING NOT FOUND":GOTO22350
22420 AA=XS
22440 IFZW$(3)=""GOTO22470
22450 AB=VAL(ZW$(3))
22460 GOTO22540
22470 IFZW$(4)<>""THENGOTO22500
```

```
22480 AB=9999
22490 GOTO22540
22500 XD$=ZW$(4):GOSUB5000
22510 IFXM<>0ORXL<>-1GOTO22530
22520 XB$="END STRING NOT FOUND":GOTO22350
22530 AB=XS
22540 XB$="DISPLAY(D) OR PRINT(P)?":XB=1:GOSUB11000:IFXX=1GOTO22950
22550 AC$=XC$
22560 IFAC$<>"D"ANDAC$<>"P"GOTO 22540
22570 IFAC$="D"GOTO22820
22590 XB$="CURRENT FORMAT (C) OR NEW (N)?":XB=1:GOSUB11000:IFXX=1GOTO22950
22600 IFXC$<>"C"ANDXC$<>"N"GOTO22590
22610 IFXC$="C"GOTO22830
22630 XB$="SUPPRESS NEW PAGE, Y OR N?":XB=2:GOSUB11000
22640 IFXC$="Y"THENZM=-1ELSEZM=50
22650 ZP=55:ZQ=6:ZP$(0)="DISPLAY/PRINT FORMAT ITEMS"
22660 ZP$(1)="0=NEW LINE":ZP$(2)="1-N=FIELD N"
22670 ZP$(3)="-M=TAB TO POSITION M":ZP$(4)="-1=NEW PAGE"
22680 ZP$(5)="-2=PRINT REPORT COUNTER XN":ZP$(6)="-3=END ITEM DEFINITION"
22690 ZR(1)=0:ZR(2)=0:ZR(3)=0:ZR(4)=0:ZR(5)=0:ZR(6)=0:GOSUB8000
22700 XP(0)=0
22710 AI=1
22720 XB$="ITEM TYPE?:":XB=0:GOSUB11000:IFXX=1GOTO22950
22730 IFXC>-63ANDXC<=8GOTO22760
22740 XB$="INVALID ITEM TYPE - IGNORED":XB=3:GOSUB11000
22750 GOTO22720
22760 IFXC=-3GOTO22830
22770 XP(AI)=XC
22780 XP(0)=XP(0)+1
22790 AI=AI+1
22800 GOTO22720
22820 ZP$(0)="DISPLAY ENTRY"
22830 IFAC$="D"THENGOSUB29000
22840 XU=0:XS=AA:XT=0:GOSUB5500
22850 XS=-1:XT=1
22860 IFXL=-1GOTO22950
22870 XY$=XA$(XL):GOSUB2500
22880 XN=XU-1
22890 IFAC$="D"THENGOSUB9000ELSEGOSUB7500
22900 IFINKEY$<>""GOTO22950
22910 IFXU>=ABGOTO22950
22920 GOSUB5500
22930 PRINT@YL,XU-2;"    ";
22940 GOTO22860
22950 YS=0
22960 FORAI=1TO900:NEXTAI
22970 RETURN
23000 GOTO23050'MFSRCH
23050 ZP=55:ZQ=1:ZP$(0)="SEARCH":ZP$(1)="SEARCH STRING:"
23060 ZR(1)=30:GOSUB8000
23080 GOSUB8500:IFXX=1GOTO23270
23090 IFZW$(1)=""GOTO23270
23100 AW$=ZW$(1)
23110 ZP$(0)="SEARCH"
```

```
23120 GOSUB29000
23140 XU=0
23150 XS=-1:XT=1:GOSUB5500
23160 IFINKEY$<>""GOTO23270
23170 IFXL=-1GOTO23270
23180 XW$=AW$
23190 XZ$=XA$(XL):GOSUB1000
23200 IFXW=-1GOTO23150
23220 PRINT@YL,XU-1;"    ";
23230 XY$=XZ$:GOSUB2500
23240 GOSUB9000
23250 XB$="CONTINUE?":XB=2:GOSUB11000:IFXX=1GOTO23270
23260 IFXC$="Y"THENGOTO23150
23270 RETURN
23500 GOTO23550'MFSEC
23550 ZP=55:ZQ=1:ZP$(0)="SECONDARY SORT":ZP$(1)="SORT ON FIELD # :"
23560 ZR(1)=1:GOSUB8000
23570 GOSUB8500:IFXX=1GOTO23620
23580 YT=VAL(ZW$(1))
23590 IFYT<1ORYT>8GOTO23550
23610 GOSUB6000
23620 RETURN
24000 GOTO24050'MFLOAD
24050 ZP=55:ZQ=3:ZP$(0)="LOAD FILE"
24060 ZP$(1)="LOAD FROM CASSETTE (C) OR DISK (D)?"
24070 ZP$(2)="DISK FILENAME?":ZP$(3)="INITIALIZE (I) OR MERGE (M)?"
24080 ZR(1)=1:ZR(2)=15:ZR(3)=1:GOSUB8000
24100 GOSUB8500:IFXX=1GOTO24150
24110 IFZW$(1)<>"C"ANDZW$(1)<>"D"GOTO24050
24120 IFZW$(3)<>"I"ANDZW$(3)<>"M"GOTO24050
24140 GOSUB12000
24150 RETURN
24500 GOTO24550'MFSAVE
24550 ZP=55:ZQ=2:ZP$(0)="SAVE FILE"
24560 ZP$(1)="SAVE ON CASSETTE (C) OR DISK (D)?"
24570 ZP$(2)="DISK FILENAME?":ZR(1)=1:ZR(2)=15:GOSUB8000
24590 GOSUB8500:IFXX=1GOTO24630
24600 IFZW$(1)<>"C"ANDZW$(1)<>"D"GOTO24550
24620 GOSUB12500
24630 RETURN
29000 GOTO29060'MLSKEL
29060 ZP=57:ZQ=8
29070 ZP$(1)="(1)  NAME1     ":ZP$(2)="(2)  NAME2     "
29080 ZP$(3)="(3)  NAME3     ":ZP$(4)="(4)  STREET    "
29090 ZP$(5)="(5)  CITY      ":ZP$(6)="(6)  STATE     "
29100 ZP$(7)="(7)  ZIP       ":ZP$(8)="(8)  REFERENCE"
29110 ZR(1)=17:ZR(2)=15:ZR(3)=15:ZR(4)=30:ZR(5)=20
29120 ZR(6)=10:ZR(7)=9:ZR(8)=10:GOSUB8000
29130 RETURN
```

Index

RADIO SHACK A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA	BELGIUM	U.K.
280-316 VICTORIA ROAD RYDALMERE, N.S.W. 2116	PARC INDUSTRIEL DE NANINNE 5140 NANINNE	BILSTON ROAD WEDNESBURY WEST MIDLANDS WS10 7JN

PRINTED IN U.S.A.